JFP: page 1 of 46. © Cambridge University Press 2012
doi:10.1017/S0956796812000056

Proofs for free

Parametricity for dependent types

JEAN-PHILIPPE BERNARDY and PATRIK JANSSON
Chalmers University of Technology & University of Gothenburg, Sweden
(*e-mail:* {bernardy,patrikj}@chalmers.se)

ROSS PATERSON
City University, London, UK
(*e-mail:* ross@soi.city.ac.uk)

Abstract

Reynolds' abstraction theorem (Reynolds, J. C. (1983) Types, abstraction and parametric polymorphism, *Inf. Process.* **83**(1), 513–523) shows how a typing judgement in System F can be translated into a relational statement (in second-order predicate logic) about inhabitants of the type. We obtain a similar result for pure type systems (PTSs): for any PTS used as a programming language, there is a PTS that can be used as a logic for parametricity. Types in the source PTS are translated to relations (expressed as types) in the target. Similarly, values of a given type are translated to proofs that the values satisfy the relational interpretation. We extend the result to inductive families. We also show that the assumption that every term satisfies the parametricity condition generated by its type is consistent with the generated logic.

1 Introduction

Types are used in many parts of computer science to keep track of different kinds of values and to keep software from going wrong. Starting from the presentation of the simply typed lambda calculus by Church (1940), we have seen a steady flow of typed languages and calculi. With increasingly rich type systems came more refined properties about well-typed terms. In his *abstraction theorem*, Reynolds (1983) defined a relational interpretation of System F types, and showed that interpretations of a well-typed term in related contexts yield related results. If a type has no free variables, the relational interpretation can thus be viewed as a *parametricity* property satisfied by all terms of that type. Almost 20 years ago Barendregt (1992) described a common framework for a large family of calculi with expressive types: Pure Type Systems (PTSs). By the Curry–Howard correspondence, the calculi in the PTS family can be seen both as programming languages and as logics. The more advanced calculi go beyond System F and include full-dependent types and support expressing datatypes.

Recent works (Takeuti 2004, personal communication; Johann & Voigtländer 2006; Neis *et al.* 2009; Vytiniotis & Weirich 2010) have developed parametricity

results for several such calculi, but not in a common framework. In this paper, we apply and extend Reynolds' (1983) idea to a large class of PTSs and provide a framework that unifies previous descriptions of parametricity and forms a basis for future studies of parametricity in specific type systems. As a by-product, we get parametricity for dependently typed languages. This paper is an extended and revised version of Bernardy *et al.* (2010). Our specific contributions are as follows:

- A concise definition of the translation of types to relations (Definition 3.9), which yields parametricity propositions for closed terms.
- A formulation (and a proof) of the abstraction theorem for PTSs (Theorem 3.12). A remarkable feature of the theorem is that the translation from types to relations and the translation from terms to proofs are unified.
- An extension of the PTS framework to capture explicit syntax (Section 4).
- An extension of the translation to inductive definitions (Section 5), and its proof of correctness.
- A formulation of an axiom schema able to internalise the abstraction theorem in a PTS. The axiom schema is proved consistent, thanks to a translation to PTS without axioms (Section 6).
- A specialisation of the general framework to constructs such as propositions, type classes, and constructor classes (Section 7).
- A demonstration by example of how to derive free theorems for (and as) dependently typed functions (Sections 3.3, 5, and 7).

Our examples use a notation close to that of Agda (Norell 2007), for greater familiarity for users of dependently typed functional programming languages. The notation takes advantage of the "implicit syntax" feature, improving the readability of examples.

2 Pure type systems

In this section we review the notion of PTS as described by Barendregt (1992, Sec. 5.2). We introduce our notation along the way, as well as our running example type systems.

Definition 2.1 (*Syntax of terms*)
A PTS is a type system over a λ-calculus with the following syntax:

$$\begin{array}{rll} \mathbb{T} = \mathbb{C} & & \text{constant} \\ \mid \mathbb{V} & & \text{variable} \\ \mid \mathbb{T}\ \mathbb{T} & & \text{application} \\ \mid \lambda \mathbb{V} : \mathbb{T}.\ \mathbb{T} & & \text{abstraction} \\ \mid \forall \mathbb{V} : \mathbb{T}.\ \mathbb{T} & & \text{dependent function space} \end{array}$$

We often write $A \to B$ for $\forall \mathsf{x} : A.\ B$ when x does not occur free in B. We use different fonts to indicate what category a meta-syntactic variable ranges over. Sans-serif roman (like x) is used for $\mathbb{V}$, fraktur (like $\mathfrak{c}$) for $\mathbb{C}$, and italics (like A) for $\mathbb{T}$. As an exception, the letters s and t are used for the subset $\mathbb{S}$ of $\mathbb{C}$ introduced in the next paragraph.

$$\frac{}{\vdash \mathsf{c} : s}\ \mathsf{c} : s \in \mathbb{A} \quad \text{AXIOM} \qquad \frac{\Gamma \vdash A : s}{\Gamma, \mathsf{x} : A \vdash \mathsf{x} : A}\ \text{START} \qquad \frac{\Gamma \vdash A : B \qquad \Gamma \vdash C : s}{\Gamma, \mathsf{x} : C \vdash A : B}\ \text{WEAKENING}$$

$$\frac{\Gamma \vdash A : s_1 \qquad \Gamma, \mathsf{x} : A \vdash B : s_2}{\Gamma \vdash (\forall \mathsf{x} : A.\ B) : s_3}\ (s_1, s_2, s_3) \in \mathbb{R}\ \text{PRODUCT} \qquad \frac{\Gamma \vdash F : (\forall \mathsf{x} : A.\ B) \qquad \Gamma \vdash a : A}{\Gamma \vdash F\ a : B[\mathsf{x} \mapsto a]}\ \text{APPLICATION}$$

$$\frac{\Gamma, \mathsf{x} : A \vdash b : B \qquad \Gamma \vdash (\forall \mathsf{x} : A.\ B) : s}{\Gamma \vdash (\lambda \mathsf{x} : A.\ b) : (\forall \mathsf{x} : A.\ B)}\ \text{ABSTRACTION} \qquad \frac{\Gamma \vdash A : B \qquad \Gamma \vdash B' : s \qquad B =_\beta B'}{\Gamma \vdash A : B'}\ \text{CONVERSION}$$

Fig. 1. Typing rules of PTS with specification $(\mathbb{S}, \mathbb{A}, \mathbb{R})$.

The typing judgement of a PTS is parametrised over a *specification* $S = (\mathbb{S}, \mathbb{A}, \mathbb{R})$, where $\mathbb{S} \subseteq \mathbb{C}$, $\mathbb{A} \subseteq \mathbb{C} \times \mathbb{S}$, and $\mathbb{R} \subseteq \mathbb{S} \times \mathbb{S} \times \mathbb{S}$. The set $\mathbb{S}$ specifies the sorts, $\mathbb{A}$ the axioms (an axiom $(\mathsf{c}, s) \in \mathbb{A}$ is often written $\mathsf{c} : s$), and $\mathbb{R}$ specifies the typing rules of the function space. A rule (s_1, s_2, s_2), where the second and third sorts coincide, is often written $s_1 \rightsquigarrow s_2$. The typing rules for a PTS are shown in Figure 1.

An attractive feature of PTSs is that the syntax for types and values is unified. It is the type of a term that tells how to interpret it (as a value, type, kind, etc.).

The λ-cube Barendregt (1992) defined a family of calculi each with $\mathbb{S} = \{\star, \Box\}$, $\mathbb{A} = \{\star : \Box\}$ and $\mathbb{R}$ a selection of rules of the form $s_1 \rightsquigarrow s_2$, for example:

- The (monomorphic) λ-calculus has $\mathbb{R}_\lambda = \{\star \rightsquigarrow \star\}$, corresponding to ordinary (value-level, non-dependent) functions.
- System F has $\mathbb{R}_\mathrm{F} = \mathbb{R}_\lambda \cup \{\Box \rightsquigarrow \star\}$, adding (impredicative) universal quantification over types (thus including functions from types to values).
- System Fω has $\mathbb{R}_{\mathrm{F}\omega} = \mathbb{R}_\mathrm{F} \cup \{\Box \rightsquigarrow \Box\}$, adding type-level functions.
- The Calculus of Constructions (CC) has $\mathbb{R}_\mathrm{CC} = \mathbb{R}_{\mathrm{F}\omega} \cup \{\star \rightsquigarrow \Box\}$, adding dependent types (functions from values to types).

Here $\star$ and $\Box$ are conventionally called the sorts of *types* and *kinds*, respectively.

Notice that F is a subsystem of Fω, which is itself a subsystem of CC. (We say that $S_1 = (\mathbb{S}_1, \mathbb{A}_1, \mathbb{R}_1)$ is a subsystem of $S_2 = (\mathbb{S}_2, \mathbb{A}_2, \mathbb{R}_2)$ when $\mathbb{S}_1 \subseteq \mathbb{S}_2$, $\mathbb{A}_1 \subseteq \mathbb{A}_2$ and $\mathbb{R}_1 \subseteq \mathbb{R}_2$.) In fact, the λ-cube is so named because the lattice of the subsystem relation between all the systems forms a cube, with CC at the top.

Sort hierarchies Difficulties with impredicativity[1] have led to the development of type systems with an infinite hierarchy of sorts. The "pure" part of such a system can be captured in the following PTS, which we name I_ω.

Definition 2.2 (I_ω)
I_ω is a PTS with this specification:

- $\mathbb{S} = \{\star_i \mid i \in \mathbb{N}\}$

[1] It is inconsistent with strong sums (Coquand 1986).

- $\mathbb{A} = \{\star_i : \star_{i+1} \mid i \in \mathbb{N}\}$
- $\mathbb{R} = \{(\star_i, \star_j, \star_{\max(i,j)}) \mid i, j \in \mathbb{N}\}$

Compared to the monomorphic λ-calculus, $\star$ has been expanded into the infinite hierarchy $\star_0, \star_1, \ldots$ In I_ω, the sort $\star_0$ (abbreviated $\star$) is called the sort of types. Type constructors, or type-level functions have type $\star \to \star$. Terms like $\star$ (representing the set of types) and $\star \to \star$ (representing the set of type constructors) have type $\star_1$ (the sort of kinds). Terms like $\star_1$ and $\star \to \star_1$ have type $\star_2$, and so on.

Although the infinite sort hierarchy was introduced to avoid impredicativity, they can in fact coexist, as Coquand (1986) has shown. For example, in the Generalised Calculus of Constructions (CC_ω) of Miquel (2001), impredicativity exists for the sort $\star$ (conventionally called the sort of *propositions*), which lies at the bottom of the hierarchy.

Definition 2.3 (CC_ω)
CC_ω is a PTS with this specification:

- $\mathbb{S} = \{\star\} \cup \{\Box_i \mid i \in \mathbb{N}\}$
- $\mathbb{A} = \{\star : \Box_0\} \cup \{\Box_i : \Box_{i+1} \mid i \in \mathbb{N}\}$
- $\mathbb{R} = \{\star \rightsquigarrow \star, \star \rightsquigarrow \Box_i, \Box_i \rightsquigarrow \star \mid i \in \mathbb{N}\} \cup \{(\Box_i, \Box_j, \Box_{\max(i,j)}) \mid i, j \in \mathbb{N}\}$

In the above definition, impredicativity is implemented by the rules of the form $\Box_i \rightsquigarrow \star$.

Both CC and I_ω are subsystems of CC_ω, with $\star_i$ in I_ω corresponding to $\Box_i$ in CC_ω. Because $\Box$ in CC corresponds to $\Box_0$ in CC_ω, we often abbreviate $\Box_0$ to $\Box$.

Many dependently typed programming languages and proof assistants are based on variants of I_ω or CC_ω, often with the addition of inductive definitions (Paulin-Mohring 1993; Dybjer 1994). Such tools include Agda (Norell 2007), Coq (The Coq development team 2010) and Epigram (McBride & McKinna 2004).

2.1 Pure type system as logical framework

Another use for PTSs is as logical frameworks: types correspond to propositions and terms to proofs. This correspondence extends to all aspects of the systems and is widely known as the Curry–Howard isomorphism. The judgement $\vdash p : P$ means that p is a *witness*, or *proof* of the proposition P. If the judgement holds (for some p), we say that P is *inhabited*.

In the logical system reading, an inhabited type corresponds to a tautology and dependent function types correspond to universal quantification. A predicate P over a type A has the type $A \to s$, for some sort s: a value a satisfies the predicate whenever the type $P\ a$ is inhabited. Similarly, binary relations between values of types A_1 and A_2 have type $A_1 \to A_2 \to s$.

For this approach to be safe, it is important that the system be *consistent*. In fact, the particular systems used here even exhibit the strong normalisation property: each witness p reduces to a normal form.

In fact, in I_ω and similarly rich type systems, one may represent both programs and logical formulae about them. In the following sections we make full use of

this property: We encode programs and parametricity statements about them in the same type system.

3 The relational interpretation

In this section we present the core contribution of this paper: The relational interpretation of a term, as a syntactic translation from terms representing programs or types (in a source PTS understood as a programming language) to terms representing proofs or relations (in a target PTS understood as a logic expressing properties of programming language terms). As we will see in Section 3.3, it is a generalisation of the classical rules given by Reynolds (1983), extended to all the constructs found in a PTS.

3.1 Preliminaries

Usual presentations of parametricity use binary relations, but for generality we abstract over the arity of relations, n, which we assume is given. We use an overbar notation to denote parts of terms being replicated n times with renaming, defined formally as follows.

Definition 3.1 (*renaming*)
The term A_i is obtained by replacing each free variable x in the term A by a variable x_i.

Definition 3.2 (*replication*)
$\overline{A}$ stands for n terms A_i, each obtained by renaming as defined above. Correspondingly, $\overline{\mathsf{x} : A}$ stands for n bindings $(\mathsf{x}_i : A_i)$. If replication is used in a binder (abstraction or dependent function space), then the binder is also replicated.

For a particular source PTS S, we shall require a target PTS S^r that includes S so that source terms can be expressed, but also sufficient sorts, axioms and rules to express the relational counterparts of the source terms. For example, we require that for each sort s in S, S^r should also include a sort $\widetilde{s}$ that will be the sort of relational propositions about terms of sort s. In many cases we use $\widetilde{s} = s$.

Below we simply list our requirements on S^r, noting where we shall need them later. The need for each of these requirements should become clear when we reach those points in our development. For a first approximation, we assume that the only constants in S are sorts. We return to the general case in Section 5.

Definition 3.3 (*reflecting system*)
A PTS $S^r = (\mathbb{S}^r, \mathbb{A}^r, \mathbb{R}^r)$ *reflects* a PTS $S = (\mathbb{S}, \mathbb{A}, \mathbb{R})$ if S is a subsystem of S^r and

1. (needed for Lemma 3.7) for each sort $s \in \mathbb{S}$,
   - $\mathbb{S}^r$ contains sorts s', $\widetilde{s}$, $\widetilde{s'}$ and $\widetilde{s''}$
   - $\mathbb{A}^r$ contains $s : s'$, $\widetilde{s} : \widetilde{s'}$ and $\widetilde{s'} : \widetilde{s''}$
   - $\mathbb{R}^r$ contains $s \rightsquigarrow \widetilde{s'}$ and $s' \rightsquigarrow \widetilde{s''}$
2. (needed for Lemma 3.8) for each axiom $s : t \in \mathbb{A}$, $\widetilde{s'} = \widetilde{t}$

3. (needed for the translation of products) for each rule $(s_1, s_2, s_3) \in \mathbb{R}$, $\mathbb{R}^r$ contains rules $(\widetilde{s_1}, \widetilde{s_2}, \widetilde{s_3})$ and $s_1 \rightsquigarrow \widetilde{s_3}$.

Example 3.4
The system CC_ω reflects each of the systems in the λ-cube, with $\widetilde{s} = s$.

Definition 3.5 (*reflective*)
We say that S is reflective if S reflects itself with $\widetilde{s} = s$.

Example 3.6
The systems I_ω and CC_ω are both reflective. Therefore, we can write programs in these systems and derive valid statements about them, within the same PTS.

3.2 From types to relations, from terms to proofs

In this section we present the relational translation of terms. We discuss the intuition behind each case of the definition before summarising them (in Definition 3.9).

The translation of a sort s forms types of n-ary relations between types of sort s. In particular, we choose to model relations between types $A_1, \ldots, A_n$ of sort s as terms of type $A_1 \to \cdots \to A_n \to \widetilde{s}$, where $\widetilde{s}$ is the sort of propositions corresponding to types of sort s. (In many cases we use $\widetilde{s} = s$.) Thus, we define the translation of s as

$$[\![s]\!] = \lambda \overline{\mathsf{x} : s}.\ \overline{\mathsf{x}} \to \widetilde{s}$$

The n lambda-abstractions over the variables $\overline{\mathsf{x}}$ name the parameter types of sort s, from which the type of relations is formed.

Lemma 3.7 ($[\![s]\!]$ *is well-typed*)
If the PTS S^r reflects the PTS S, then for each sort $s \in \mathbb{S}$ we have $\vdash [\![s]\!] : \overline{s} \to \widetilde{s}'$ in S^r.

Proof
From the requirements for a sort $s \in \mathbb{S}$ in the first part of Definition 3.3, we can infer (in S^r)

$$\dfrac{\dfrac{\dfrac{}{\vdash s : s'}}{\overline{\mathsf{x} : s} \vdash \mathsf{x}_i : s}\ \mathrm{st} \qquad \dfrac{\dfrac{}{\vdash \widetilde{s} : \widetilde{s}'} \quad \dfrac{}{\vdash s : s'}}{\overline{\mathsf{x} : s} \vdash \widetilde{s} : \widetilde{s}'}\ \mathrm{wk}}{\overline{\mathsf{x} : s} \vdash \overline{\mathsf{x}} \to \widetilde{s} : \widetilde{s}'}\ s \rightsquigarrow \widetilde{s}' \qquad \dfrac{\dfrac{}{\vdash s : s'} \quad \dfrac{}{\vdash \widetilde{s}' : \widetilde{s}''}}{\vdash \overline{s} \to \widetilde{s}' : \widetilde{s}''}\ s' \rightsquigarrow \widetilde{s}''}{\vdash (\lambda \overline{\mathsf{x} : s}.\ \overline{\mathsf{x}} \to \widetilde{s}) : \overline{s} \to \widetilde{s}'}\ \mathrm{abs} \qquad \square$$

Moreover, if two sorts are related by an axiom, their translations are related.

Lemma 3.8
If the PTS S^r reflects the PTS S and $\mathbb{A}$ contains an axiom $s : t$, then $\vdash [\![s]\!] : \widetilde{t}\ \overline{s}$ in S^r.

Proof

$$\dfrac{\dfrac{\vdots\ \text{Lemma 3.7}}{\vdash [\![s]\!] : \overline{s} \to \widetilde{s}'} \qquad \dfrac{\dfrac{\vdots\ \text{Lemma 3.7}}{\vdash (\lambda \overline{\mathsf{x} : t}.\ \overline{\mathsf{x}} \to \widetilde{t}) : \overline{t} \to \widetilde{t}'} \quad \dfrac{}{\vdash s : t}\ \mathrm{app}}{\vdash (\lambda \overline{\mathsf{x} : t}.\ \overline{\mathsf{x}} \to \widetilde{t})\ \overline{s} : \widetilde{t}'}}{\vdash [\![s]\!] : (\lambda \overline{\mathsf{x} : t}.\ \overline{\mathsf{x}} \to \widetilde{t})\ \overline{s}}\ \mathrm{conv},\ \widetilde{s}' = \widetilde{t} \qquad \square$$

Note that this proof uses the equality from the second part of Definition 3.3.

Generalising Lemma 3.8, for each type $A : s$, we wish to define a relation $[\![A]\!] : [\![s]\!]\,\overline{A}$. Type systems usually include constants that are not sorts, but as their meaning is unconstrained, we cannot expect a generic translation for them. We shall deal with such constants in Section 5.

We shall approach dependent product types through special cases. Firstly, the relation $[\![A \to B]\!]$ relates functions if they map inputs related by $[\![A]\!]$ to outputs related by $[\![B]\!]$:

$$[\![A \to B]\!] = \lambda \overline{\mathsf{f} : (A \to B)}.\ \forall \overline{\mathsf{x} : A}.\ [\![A]\!]\ \overline{\mathsf{x}} \to [\![B]\!]\ (\overline{\mathsf{f}\ \mathsf{x}})$$

Secondly, the relation $[\![\forall \mathsf{x} : s.\ B]\!]$ relates polymorphic terms if their instances at related types are related:

$$\begin{aligned}[\![\forall \mathsf{x} : s.\ B]\!] &= \lambda \overline{\mathsf{f} : (\forall \mathsf{x} : s.\ B)}.\ \forall \overline{\mathsf{x} : s}.\ \forall \mathsf{x}_\mathsf{R} : \overline{\mathsf{x}} \to \widetilde{s}.\ [\![B]\!]\ (\overline{\mathsf{f}\ \mathsf{x}}) \\ &= \lambda \overline{\mathsf{f} : (\forall \mathsf{x} : s.\ B)}.\ \forall \overline{\mathsf{x} : s}.\ \forall \mathsf{x}_\mathsf{R} : [\![s]\!]\ \overline{\mathsf{x}}.\ [\![B]\!]\ (\overline{\mathsf{f}\ \mathsf{x}})\end{aligned}$$

Both of these forms are special cases of the general translation of products as follows:

$$[\![\forall \mathsf{x} : A.\ B]\!] = \lambda \overline{\mathsf{f} : (\forall \mathsf{x} : A.\ B)}.\ \forall \overline{\mathsf{x} : A}.\ \forall \mathsf{x}_\mathsf{R} : [\![A]\!]\ \overline{\mathsf{x}}.\ [\![B]\!]\ (\overline{\mathsf{f}\ \mathsf{x}})$$

Products are also types, and hence are also translated to relations via lambda-abstractions over n functions $\overline{\mathsf{f}}$. The right-hand side of the product ends the description of how the functions f must be related by requiring that the result of applying $\overline{\mathsf{f}}$ to $\overline{\mathsf{x}}$ be related by the translation of B.

In the above translation, if the source product $\forall \mathsf{x} : A.\ B$ is formed with the rule (s_1, s_2, s_3), then $[\![A]\!]\ \overline{\mathsf{x}}$ has sort $\widetilde{s_1}$, while $[\![B]\!]\ (\overline{\mathsf{f}\ \mathsf{x}})$ has sort $\widetilde{s_2}$. Thus, S^r requires the rule $(\widetilde{s_1}, \widetilde{s_2}, \widetilde{s_3})$ in order to form the inner product on the right-hand side. Similarly, the outer product requires the rule $s_1 \rightsquigarrow \widetilde{s_3}$. These rules are those of the third part of Definition 3.3.

The translation of applications and abstraction mirrors the translation of product types at the value level: one argument is mapped to n arguments and a relation argument,

$$\begin{aligned}[\![F\ a]\!] &= [\![F]\!]\ \overline{a}\ [\![a]\!] \\ [\![\lambda \mathsf{x} : A.\ b]\!] &= \lambda \overline{\mathsf{x} : A}.\ \lambda \mathsf{x}_\mathsf{R} : [\![A]\!]\ \overline{\mathsf{x}}.\ [\![b]\!]\end{aligned}$$

The translation maintains the invariant that for each free variable in the input x, the output has $n + 1$ free variables, $\mathsf{x}_1, \ldots, \mathsf{x}_n$ and x_R, where x_R witnesses that $\mathsf{x}_1, \ldots, \mathsf{x}_n$ are related. Hence, a variable x can be translated to x_R.

The translation of terms is summed up in the following definition, which gives the mapping $[\![\_]\!]$ from terms of a PTS S to terms of a possibly extended PTS S^r as follows.

Definition 3.9 (*parametricity translation from types to relations*)

$$\begin{aligned}
[\![s]\!] &= \lambda \overline{\mathsf{x} : s}.\ \overline{\mathsf{x}} \to \widetilde{s} \\
[\![\mathsf{x}]\!] &= \mathsf{x}_\mathsf{R} \\
[\![\forall \mathsf{x} : A.\ B]\!] &= \lambda \overline{\mathsf{f} : (\forall \mathsf{x} : A.\ B)}.\ \forall \overline{\mathsf{x} : A}.\ \forall \mathsf{x}_\mathsf{R} : [\![A]\!]\ \overline{\mathsf{x}}.\ [\![B]\!]\ (\overline{\mathsf{f}\ \mathsf{x}}) \\
[\![F\ a]\!] &= [\![F]\!]\ \overline{a}\ [\![a]\!] \\
[\![\lambda \mathsf{x} : A.\ b]\!] &= \lambda \overline{\mathsf{x} : A}.\ \lambda \mathsf{x}_\mathsf{R} : [\![A]\!]\ \overline{\mathsf{x}}.\ [\![b]\!]
\end{aligned}$$

The replication of variables carries on to contexts.

Definition 3.10 (*parametricity translation for contexts*)

$$\begin{aligned}
[\![-]\!] &= - \\
[\![\Gamma,\ \mathsf{x} : A]\!] &= [\![\Gamma]\!],\ \overline{\mathsf{x} : A},\ \mathsf{x}_\mathsf{R} : [\![A]\!]\ \overline{\mathsf{x}}
\end{aligned}$$

Note that each tuple $\overline{\mathsf{x} : A}$ in the translated context must satisfy the relation $[\![A]\!]$, as witnessed by x_R. Thus, one may interpret $[\![\Gamma]\!]$ as n related environments; and $\overline{A}$ as n interpretations of A, each one in a different environment.

Lemma 3.11 (*translation preserves β-reduction*)

$$A \longrightarrow_\beta^* A' \implies [\![A]\!] \longrightarrow_\beta^* [\![A']\!]$$

Proof sketch
The proof proceeds by induction on the derivation of $A \longrightarrow_\beta^* A'$. The interesting case is where the term A is a β-redex $(\lambda \mathsf{x} : B.\ C)\ b$. That case relies on the way $[\![\_]\!]$ interacts with substitution:

$$[\![b[\mathsf{x} \mapsto C]]\!] = [\![b]\!][\overline{\mathsf{x}} \mapsto \overline{C}][\mathsf{x}_\mathsf{R} \mapsto [\![C]\!]]$$

The remaining cases are congruences. □

We can then state our main result.

Theorem 3.12 (*abstraction*)
If the PTS S^r reflects the PTS S,

$$\Gamma \vdash_S A : B \implies [\![\Gamma]\!] \vdash_{S^r} [\![A]\!] : [\![B]\!]\ \overline{A}$$

Proof
By induction on the derivation of $\Gamma \vdash_S A : B$. Each typing rule in the derivation of the source judgement can be translated to a portion of the derivation tree of the target. The START case is a consequence of the invariant that a relational witness is always introduced in the context when a variable is bound in the source term. The cases of ABSTRACTION and APPLICATION stem from the fact that their respective translations follow the pattern of the translation of the product. The

PRODUCT case uses the fact that types are translated to relations (in $[\![s]\!]$), and imposes constraints on the structure of the target PTS (see Definition 3.3). In the AXIOM case, we rely on the "types-to-relations" principle at two different levels, and further conditions are imposed on the target PTS. More details of the proof are given in Appendix A.1. □

The above theorem can be read in two ways. A direct reading is as a typing judgement about translated terms: if A has type B, then $[\![A]\!]$ has type $[\![B]\!]\ \overline{A}$. The more fruitful reading is as an abstraction theorem for PTSs: if A has type B in environment Γ, then n interpretations $\overline{A}$ in related environments $[\![\Gamma]\!]$ are related by $[\![B]\!]$. Further, $[\![A]\!]$ is a witness of this proposition *within the type system*. In particular, closed terms are related to themselves: $\vdash A : B \implies \vdash [\![A]\!] : [\![B]\!]\ A \ldots A$.

3.3 Examples: the λ-cube

In this section we show that $[\![\_]\!]$ specialises to the rules given by Reynolds (1983) to read a System F type as a relation. Having shown that our framework can explain parametricity theorems for System F types, we move on to progressively higher order constructs. In these examples, the binary version of parametricity is used (arity $n = 2$). Using Definition 3.3 one can verify that the following system reflects System F.

- $\mathbb{S} = \{\star, \Box, \Box_1, \widetilde{\star}, \widetilde{\Box}, \widetilde{\Box}_1, \widetilde{\Box}_2\}$
- $\mathbb{A} = \{\star : \Box, \Box : \Box_1, \widetilde{\star} : \widetilde{\Box}, \widetilde{\Box} : \widetilde{\Box_1}, \widetilde{\Box_1} : \widetilde{\Box_2}\}$
- $\mathbb{R} = \{\star \rightsquigarrow \star, \Box \rightsquigarrow \star, \star \rightsquigarrow \widetilde{\Box}, \Box \rightsquigarrow \widetilde{\Box}_1, \Box_1 \rightsquigarrow \widetilde{\Box}_2, \widetilde{\star} \rightsquigarrow \widetilde{\star}, \widetilde{\Box} \rightsquigarrow \widetilde{\star}\}$

Indeed, examination of the structure of the PTS reveals that it corresponds to a second-order logic with typed individuals, studied multiple times in the literature with slight variations, for example by Plotkin & Abadi (1993) or Wadler (2007). In the PTS form, the sort $\widetilde{\star}$ is the sort of propositions. The sort $\widetilde{\Box}$ is inhabited by the type of propositions ($\widetilde{\star}$), the type of predicates ($\tau \to \widetilde{\star}$), and in general types of relations ($\tau_1 \to \cdots \to \tau_n \to \widetilde{\star}$). The sorts $\Box_1$ and $\Box_2$ come from the need to type unimportant higher level redexes created by our translation, and correspond to the sorts with the same name in CC_ω. The product formation rules can be understood as follows:

- $\widetilde{\star} \rightsquigarrow \widetilde{\star}$ allows to build implication between propositions;
- $\star \rightsquigarrow \widetilde{\star}$ allows to quantify over programs in propositions;
- $\Box \rightsquigarrow \widetilde{\star}$ allows to quantify over types in propositions;
- $\star \rightsquigarrow \widetilde{\Box}$ is used to build types of predicates depending on programs;
- $\widetilde{\Box} \rightsquigarrow \widetilde{\star}$ allows to quantify over predicates in propositions.
- The other rules, involving $\Box_1$ and $\Box_2$ come from the need to type higher level relation-membership redexes.

Types to relations Note that by definition,

$$[\![\star]\!]\ \mathsf{T}_1\ \mathsf{T}_2 \ = \ \mathsf{T}_1 \to \mathsf{T}_2 \to \widetilde{\star}$$

Here we use $\widetilde{\star}$ on the right side as the sort of propositions. This means that types are translated to relations (as desired).

Function types Applying our translation to a closed non-dependent function type, we get:

$$\begin{aligned}&[\![A \to B]\!] : [\![\star]\!]\ (A \to B)\ (A \to B)\\ &[\![A \to B]\!]\ \mathsf{f}_1\ \mathsf{f}_2 \;=\; \forall \mathsf{a}_1 : A.\ \forall \mathsf{a}_2 : A.\ [\![A]\!]\ \mathsf{a}_1\ \mathsf{a}_2 \to [\![B]\!]\ (\mathsf{f}_1\ \mathsf{a}_1)\ (\mathsf{f}_2\ \mathsf{a}_2)\end{aligned}$$

That is, functions are related iff they take related arguments into related outputs.

Type schemes System F includes universal quantification of the form $\forall \mathsf{A} : \star.\ B$. Applying $[\![\_]\!]$ to this type expression yields:

$$\begin{aligned}&[\![\forall \mathsf{A} : \star.\ B]\!] : [\![\star]\!]\ (\forall \mathsf{A} : \star.\ B)\ (\forall \mathsf{A} : \star.\ B)\\ &[\![\forall \mathsf{A} : \star.\ B]\!]\ \mathsf{g}_1\ \mathsf{g}_2 \;=\; \forall \mathsf{A}_1 : \star.\ \forall \mathsf{A}_2 : \star.\ \forall \mathsf{A}_\mathsf{R} : [\![\star]\!]\ \mathsf{A}_1\ \mathsf{A}_2.\ [\![B]\!]\ (\mathsf{g}_1\ \mathsf{A}_1)\ (\mathsf{g}_2\ \mathsf{A}_2)\end{aligned}$$

In words, polymorphic values are related iff instances at related types are related. Note that because A may occur free in B, the variables A_1, A_2, and A_R may occur free in $[\![B]\!]$.

Type constructors With the addition of the rule $\Box \rightsquigarrow \Box$, one can construct terms of type $\star \to \star$, which are sometimes known as type constructors, type formers, or type-level functions. As Voigtländer (2009) remarks, extending the Reynolds-style parametricity to support-type constructors appears to be a folklore. Such folklore can be precisely justified by our framework by applying $[\![\_]\!]$ to obtain the relational counterpart of type constructors:

$$\begin{aligned}&[\![\star \to \star]\!] : [\![\Box]\!]\ (\star \to \star)\ (\star \to \star)\\ &[\![\star \to \star]\!]\ \mathsf{F}_1\ \mathsf{F}_2 \;=\; \forall \mathsf{A}_1 : \star.\ \forall \mathsf{A}_2 : \star.\ [\![\star]\!]\ \mathsf{A}_1\ \mathsf{A}_2 \to [\![\star]\!]\ (\mathsf{F}_1\ \mathsf{A}_1)\ (\mathsf{F}_2\ \mathsf{A}_2)\end{aligned}$$

That is, a term of type $[\![\star \to \star]\!]\ \mathsf{F}_1\ \mathsf{F}_2$ is a (polymorphic) function converting a relation between any types A_1 and A_2 to a relation between $\mathsf{F}_1\ \mathsf{A}_1$ and $\mathsf{F}_2\ \mathsf{A}_2$, a *relational action*. For the target system to accept the above, the rules $\Box \rightsquigarrow \widetilde{\Box}$ and $\widetilde{\Box} \rightsquigarrow \widetilde{\Box}$ must also be added there.

Dependent functions In a system with the rule $\star \rightsquigarrow \Box$, value variables may occur in dependent function types like $\forall \mathsf{x} : A.\ B$, which we translate as follows:

$$\begin{aligned}&[\![\forall \mathsf{x} : A.\ B]\!] : [\![\star]\!]\ (\forall \mathsf{x} : A.\ B)\ (\forall \mathsf{x} : A.\ B)\\ &[\![\forall \mathsf{x} : A.\ B]\!]\ \mathsf{f}_1\ \mathsf{f}_2 \;=\; \forall \mathsf{x}_1 : A.\ \forall \mathsf{x}_2 : A.\ \forall \mathsf{x}_\mathsf{R} : [\![A]\!]\ \mathsf{x}_1\ \mathsf{x}_2.\ [\![B]\!]\ (\mathsf{f}_1\ \mathsf{x}_1)\ (\mathsf{f}_2\ \mathsf{x}_2)\end{aligned}$$

Here, the target system is extended with the rule $\widetilde{\star} \rightsquigarrow \widetilde{\Box}$. The rule $\star \rightsquigarrow \widetilde{\Box}$ is also required, but is already in the system, as it is required by the source axiom $\star : \Box$ as well.

Proof terms We have used $[\![\_]\!]$ to turn types into relations, but we can also use it to turn terms into proofs of abstraction properties. As a simple example, the relation corresponding to the type $\mathsf{T} = (\mathsf{A} : \star) \to \mathsf{A} \to \mathsf{A}$, namely

$$\begin{aligned}[\![\mathsf{T}]\!]\ \mathsf{f}_1\ \mathsf{f}_2 \ = \ & \forall \mathsf{A}_1 : \star.\ \forall \mathsf{A}_2 : \star.\ \forall \mathsf{A}_\mathsf{R} : [\![\star]\!]\ \mathsf{A}_1\ \mathsf{A}_2. \\ & \forall \mathsf{x}_1 : \mathsf{A}_1.\ \forall \mathsf{x}_2 : \mathsf{A}_2.\ \mathsf{A}_\mathsf{R}\ \mathsf{x}_1\ \mathsf{x}_2 \to \mathsf{A}_\mathsf{R}\ (\mathsf{f}_1\ \mathsf{A}_1\ \mathsf{x}_1)\ (\mathsf{f}_2\ \mathsf{A}_2\ \mathsf{x}_2)\end{aligned}$$

states that functions of type T map related inputs to related outputs, for any relation. From a term $\mathsf{id} = \lambda A : *.\ \lambda x : \mathsf{A}.\ \mathsf{x}$ of this type, by the abstraction theorem we obtain a term $[\![\mathsf{id}]\!] : [\![\mathsf{T}]\!]\ \mathsf{id}\ \mathsf{id}$, that is a proof of the abstraction property:

$$[\![\mathsf{id}]\!]\ \mathsf{A}_1\ \mathsf{A}_2\ \mathsf{A}_\mathsf{R}\ \mathsf{x}_1\ \mathsf{x}_2\ \mathsf{x}_\mathsf{R} \ = \ \mathsf{x}_\mathsf{R}$$

We return to proof terms in Section 5.3 after introducing datatypes.

4 Coloured pure type systems

In this section we introduce the notion of coloured pure type system (CPTS), which is an extension of PTS as described in Section 2. Colours capture the fact that various flavours of quantification use different syntax. We use colours to improve the clarity of the relational translation as well as that of examples.

4.1 Explicit Syntax: Coloured Pure Type Systems

The complete uniformity of syntax characteristic of classical presentations of the PTS framework often obscures the structure of ideas expressed within particular PTS, and our relational interpretation of terms in no exception. While mere PTSs are sufficient for most of the technical results of this paper, the structure of the relational interpretation appears more clearly when various flavours of quantification are properly identified.

Explicit syntax in PTSs is not novel: Many systems usually presented as PTSs still use different syntax for various forms of quantifications. For example, traditional presentations of System F use a different syntax for the quantification over individuals (rule $\star \rightsquigarrow \star$) than for the quantification over types (rule $\Box \rightsquigarrow \star$). A common practice is to use the symbols $\forall$ and Λ for quantification and abstraction over types, and $\to$ and λ for individuals. In addition, brackets are sometimes used to mark type application. While the flavour of quantification can always be recovered from a type derivation, the advantage of explicit syntax is that it is possible to identify which flavour is used merely by looking at the term. Moreover, a type-derivation tree might not be available.

In this paper we want to give a relational interpretation of terms parameterised over any PTS, and retain the possibility to keep syntax annotations. This is exactly the purpose of CPTSs: to capture explicit syntax in a parametrised way. A colour annotation is added to the syntax of application, abstraction, and product, and a colour component is added to $\mathbb{R}$. A rule (k, s_1, s_2, s_2) is often written $s_1 \overset{k}{\rightsquigarrow} s_2$. Note that a single colour may be used in multiple rules. (In the electronics version of this document, colours are sometimes rendered visually.) The corresponding typing rules

$$\begin{array}{rll} \mathbb{T}_{\mathbb{K}} = \mathbb{C} & & \text{constant} \\ \quad \mid \mathbb{V} & & \text{variable} \\ \quad \mid \mathbb{T} \bullet_{\mathbb{K}} \mathbb{T} & & \text{application} \\ \quad \mid \lambda^{\mathbb{K}} \mathbb{V} : \mathbb{T}.\ \mathbb{T} & & \text{abstraction} \\ \quad \mid \forall^{\mathbb{K}} \mathbb{V} : \mathbb{T}.\ \mathbb{T} & & \text{dependent function space} \end{array}$$

$$\frac{\Gamma \vdash A : s_1 \qquad \Gamma, \mathsf{x} : A \vdash B : s_2}{\Gamma \vdash (\forall^k \mathsf{x} : A.\ B) : s_3}\ (k, s_1, s_2, s_3) \in \mathbb{R} \quad \text{PRODUCT}$$

$$\frac{\Gamma \vdash F : (\forall^k \mathsf{x} : A.\ B) \qquad \Gamma \vdash a : A}{\Gamma \vdash F \bullet_k a : B[\mathsf{x} \mapsto a]} \quad \text{APPLICATION}$$

$$\frac{\Gamma, \mathsf{x} : A \vdash b : B \qquad \Gamma \vdash (\forall^k \mathsf{x} : A.\ B) : s}{\Gamma \vdash (\lambda^k \mathsf{x} : A.\ b) : (\forall^k \mathsf{x} : A.\ B)} \quad \text{ABSTRACTION}$$

Fig. 2. CPTS syntax for the set of colours $\mathbb{K}$, and typing rules of the CPTS with specification $(\mathbb{K}, \mathbb{S}, \mathbb{A}, \mathbb{R})$. The only change with respect to the standard PTS definition is the addition of colour annotations in product, application, and abstraction.

ensure that the colours are matched (Figure 2). Erasure of colour yields a plain (monochrome) PTS; and erasure of colour in a valid coloured derivation tree yields a valid derivation tree in the monochrome PTS. Therefore, useful properties of PTSs (such as subject reduction, substitution, etc.) are retained in CPTSs.

4.2 Relational translation, with colour

We can modify our translation to use colours to distinguish the two kinds of arguments it introduces so that a single product of colour k is translated to two kinds of products, n of colour k_i, which introduces n terms $\overline{\mathsf{x}}$ of type A_i, and one of colour k_r, which forces them ($\overline{\mathsf{x}}$) to be related by the translation of A. (Memory aid: i stands for individual and r for relation.)

$$\begin{aligned} [\![\forall^k \mathsf{x} : A.\ B]\!] &= \lambda \overline{\mathsf{f} : (\forall^k \mathsf{x} : A.\ B)}.\ \forall^{k_i} \overline{\mathsf{x} : A}.\ \forall^{k_r} \mathsf{x}_{\mathsf{R}} : [\![A]\!]\ \overline{\mathsf{x}}.\ [\![B]\!]\ (\overline{\mathsf{f} \bullet_k \mathsf{x}}) \\ [\![F \bullet_k a]\!] &= [\![F]\!] \bullet_{k_i} \overline{a} \bullet_{k_r} [\![a]\!] \\ [\![\lambda^k \mathsf{x} : A.\ b]\!] &= \lambda^{k_i} \overline{\mathsf{x} : A}.\ \lambda^{k_r} \mathsf{x}_{\mathsf{R}} : [\![A]\!]\ \overline{\mathsf{x}}.\ [\![b]\!] \end{aligned}$$

We use a special, new colour (named 0 below) for the formation of relations that interpret types. Since this colour is used very many times, leave out the annotation for it. Using this convention, the translation of a sort s looks exactly the same when colours are used as in the monochrome case:

$$[\![s]\!] = \lambda \overline{\mathsf{x} : s}.\ \overline{\mathsf{x}} \to \widetilde{s}$$

The colour 0 was already used in the first set of equations given in this section, for example, in the abstraction over f, or in the applications of $[\![A]\!]$. Thanks to colours, it becomes syntactically obvious that the abstraction over f creates a relation (interpreting a type), whereas the abstraction over x does not.

The definition of reflecting system is correspondingly extended to CPTSs as follows.

Definition 4.1 (*reflecting system, with colour*)
A CPTS $S^r = (\mathbb{K}^r, \mathbb{S}^r, \mathbb{A}^r, \mathbb{R}^r)$ *reflects* a CPTS $S = (\mathbb{K}, \mathbb{S}, \mathbb{A}, \mathbb{R})$ if S is a subsystem of S^r and

1. there is a colour $0 \in \mathbb{K}^r$, used for relation construction. Annotations for this colour are consistently omitted in the remainder of the section,
2. there are two functions $\__i$ and $\__r$ from $\mathbb{K}$ to $\mathbb{K}^r$,
3. for each sort $s \in \mathbb{S}$,
   - $\mathbb{S}^r$ contains sorts s', $\widetilde{s}$, $\widetilde{s'}$ and $\widetilde{s''}$
   - $\mathbb{A}^r$ contains $s : s'$, $\widetilde{s} : \widetilde{s'}$ and $\widetilde{s'} : \widetilde{s''}$
   - $\mathbb{R}^r$ contains $s \rightsquigarrow \widetilde{s'}$ and $s' \rightsquigarrow \widetilde{s''}$
4. for each axiom $s : t \in \mathbb{A}$, $\widetilde{s'} = \widetilde{t}$,
5. for each rule $(k, s_1, s_2, s_3) \in \mathbb{R}$, $\mathbb{R}^r$ contains rules $(k_r, \widetilde{s_1}, \widetilde{s_2}, \widetilde{s_3})$ and $s_1 \overset{k_i}{\rightsquigarrow} \widetilde{s_3}$.

Remark 4.2
The above definition is intuitively justified as follows:

1. The colour 0 is used for formation of parametricity relations.
2. For each colour $k \in \mathbb{K}$,
   - the colour k_i is used for universal quantification over individuals in logical formulas;
   - the colour k_r is used for quantifications over propositions in the target system.
3. For each sort s, the sort $\widetilde{s}$ is the sort of parametricity propositions about types in s, and must exist in $\mathbb{S}^r$. One can see $\widetilde{\_}$ as a function from $\mathbb{S}$ to $\mathbb{S}^r$.
   For each input sort, the relational interpretation creates redexes, which check relation membership. This requires
   - each input sort s to be typeable (i.e. inhabit another sort s' – in the above definition we consistently use s' for a sort that s inhabits);
   - two extra sorts in the target system ($\widetilde{s'}, \widetilde{s''}$) on top of $\widetilde{s}$;
   - rules to allow for the formation of relations.
4. The following two relations between sorts must commute:
   - axiomatic inhabitation ($\mathbb{A}$);
   - correspondence between a sort of types and a sort of relational propositions ($\widetilde{\_}$).

   This point is not a strict requirement for the abstraction theorem to hold. However, we found that without this requirement, the structure of the target system is too unconstrained to make intuitive sense of it.
5. For each type-formation rule of the input system, there is
   - a formation rule for quantification over individuals;
   - a formation rule for relational-propositions, exactly mirroring that of the input system.

4.3 Coloured examples

A colour for naive set-theory Earlier in this paper, we have outlined how PTSs can be used to represent concepts like propositions and proofs. One may want to use special syntax for PTS constructs when the propositions-as-types interpretation is intended: even though propositions and types are syntactically unified in PTSs, it can be useful to make the intent explicit. Therefore, a special colour might be reserved for the purpose of expressing logical formulae in some CPTSs. A possible choice of concrete syntax is the following, reminiscent of naive set theory.

$$\begin{aligned}\mathbb{T}_{\text{logic}} = \ldots \\ \mid\ & \mathbb{T} \in \mathbb{T} && \text{(reverse application)} \\ \mid\ & \{\mathbb{V} : \mathbb{T} \mid \mathbb{T}\} && \text{(abstraction)} \\ \mid\ & \forall \mathbb{V} : \mathbb{T}.\ \mathbb{T} && \text{(quantification)}\end{aligned}$$

Classic presentations of parametricity use similar syntax, and by simply choosing this syntax for some of the colours in our PTSs, we are able to underline the similarity of our framework with previous work (Section 3.3).

A colour for implicit syntax Many proof assistants and dependently typed programming languages (including Agda, Coq, and LEGO) provide the so-called "implicit" syntax. The rationale for the feature is that, in the presence of precise type information, some parts of terms (applications or abstractions) can be fully inferred by the type-checker. In such cases, the user might want to actually leave out such parts of the terms. It is convenient to do so by marking certain quantifications as "implicit". Then the presence of the corresponding applications and abstractions can be inferred by the type-checker.

Such marking can be modelled by a two-colour PTS: one colour for regular syntax, and another for "implicit syntax". (Typically, every rule is available in both the colours.) The syntax of CPTSs does not allow for omission of terms though, so it can be used only for terms whose omitted parts have been filled in by the type-checker. Miquel (2001, section 1.3.2) gives a detailed overview of two-colour PTSs used for implicit syntax.

4.4 Implicit syntax

In the following sections, our examples are written using the Agda syntax, and take advantage of the implicit syntax feature. The following colour-set is used: $\mathbb{K} = \{e, i\}$ (e = explicit colour; i = implicit colour). Rather than using colour annotations, the following (Agda-style) concrete syntax is used.

Definition 4.3 (*Agda-style syntax for two-colour PTS*)

$$\begin{aligned}\mathbb{T} = \ & \mathbb{C} && \text{constant} \\ \mid\ & \mathbb{V} && \text{variable} \\ \mid\ & \mathbb{T}\ \mathbb{T} && \text{application} \\ \mid\ & \lambda \mathbb{V} : \mathbb{T}.\ \mathbb{T} && \text{abstraction} \\ \mid\ & (\mathbb{V} : \mathbb{T}) \to \mathbb{T} && \text{dependent function space}\end{aligned}$$

$$
\begin{array}{ll}
\mid \mathbb{T}\,\{\mathbb{T}\} & \text{implicit application} \\
\mid \lambda\{\mathbb{V}:\mathbb{T}\}.\ \mathbb{T} & \text{implicit abstraction} \\
\mid \{\mathbb{V}:\mathbb{T}\} \to \mathbb{T} & \text{implicit dependent function space}
\end{array}
$$

In addition, implicit abstraction and application may be left out when the context allows it (we do not formalise this notion). We use the following colour-mappings:

$$
\begin{array}{ll}
0 \mapsto e & \\
i_r \mapsto e & i_i \mapsto i \\
e_r \mapsto e & e_i \mapsto i
\end{array}
$$

The instantiation of $[\![\_]\!]$ (Definition 3.9) to the above mapping yields the following translation if written with the Agda-style syntax.

Example 4.4 (*translation from types to relations, specialised to implicit arguments*)

$$
\begin{aligned}
[\![s]\!] &= \lambda\overline{\mathsf{x}:s}.\ \overline{\mathsf{x}} \to \widetilde{s} \\
[\![\mathsf{x}]\!] &= \mathsf{x}_\mathsf{R} \\
[\![(\mathsf{x}:A) \to B]\!] &= \lambda\overline{\mathsf{f}:((\mathsf{x}:A) \to B)}.\ \{\overline{\mathsf{x}:A}\} \to (\mathsf{x}_\mathsf{R}:[\![A]\!]\ \overline{\mathsf{x}}) \to [\![B]\!]\ (\overline{\mathsf{f}\ \mathsf{x}}) \\
[\![F\ a]\!] &= [\![F]\!]\ \{\overline{a}\}\ [\![a]\!] \\
[\![\lambda\mathsf{x}:A.\ b]\!] &= \lambda\{\overline{\mathsf{x}:A}\}.\ \lambda\mathsf{x}_\mathsf{R}:[\![A]\!]\ \overline{\mathsf{x}}.\ [\![b]\!] \\
[\![\{\mathsf{x}:A\} \to B]\!] &= \lambda\overline{\mathsf{f}:(\{\mathsf{x}:A\} \to B)}.\ \{\overline{\mathsf{x}:A}\} \to (\mathsf{x}_\mathsf{R}:[\![A]\!]\ \overline{\mathsf{x}}) \to [\![B]\!]\ (\overline{\mathsf{f}\ \{\mathsf{x}\}}) \\
[\![F\ \{a\}]\!] &= [\![F]\!]\ \{\overline{a}\}\ [\![a]\!] \\
[\![\lambda\{\mathsf{x}:A\}.\ b]\!] &= \lambda\{\overline{\mathsf{x}:A}\}.\ \lambda\mathsf{x}_\mathsf{R}:[\![A]\!]\ \overline{\mathsf{x}}.\ [\![b]\!]
\end{aligned}
$$

The usage of implicit syntax in the translation is not innocent: It is carefully designed to take advantage of the type-inference mechanism to allow shorter expressions of translations. For example, $[\![\mathsf{id}]\!]$, generated from $\mathsf{id} : \mathsf{T}$ can now hide four out of six abstractions:

$$[\![\mathsf{id}]\!]\ \mathsf{A}_\mathsf{R}\ \mathsf{x}_\mathsf{R} = \mathsf{x}_\mathsf{R}$$

This example is typical. Indeed, we observed that for all terms A of type B, given the typing constraint $[\![A]\!] : [\![B]\!]\ \overline{A}$, arguments can be inferred at every implicit application in the expansion of $[\![A]\!]$. Likewise, every implicit abstraction is inferable and can be omitted. We have found these shortcuts to be essential to readability, as they hide much of the noise generated by the relational transformation. Therefore, we have taken advantage of inference wherever possible in the examples presented in this paper, starting from Section 5.

5 Constants and datatypes

While the above development assumes as input PTSs with $\mathbb{C} = \mathbb{S}$, it is possible to add constants to the system and retain parametricity as long as each constant is parametric. That is, for each new ("impure") axiom $\vdash_S \mathsf{c} : A$ (where c is an arbitrary

constant and A an arbitrary term, not a mere sort) we require a term $[\![\mathsf{c}]\!]$ such that the judgement $\vdash_{S^r} [\![\mathsf{c}]\!] : [\![A]\!]\ \bar{\mathsf{c}}$ holds. If the constants come with additional β-conversion rules, the translation must also preserve conversion so that Lemma 3.11 holds in the extended system: for any term A involving c, $A \longrightarrow_\beta A' \implies [\![A]\!] \longrightarrow_\beta^* [\![A']\!]$.

One source of constants in many languages is datatype definitions. In the rest of this section we investigate the implications of parametricity conditions on datatypes, and give two translation schemes for inductive families (as an extension of I_ω). In Section 5.3 we show how the term $[\![\mathsf{c}]\!]$ can be constructed from pairs and units, while in Section 5.4 we define it using another datatype definition (in which we have a constructor named $[\![\mathsf{c}]\!]$).

5.1 Parametricity and elimination

Reynolds (1983) and Wadler (1989) assume that each type constant $\mathsf{K} : \star$ is translated to the identity relation. This definition is certainly compatible with the condition required by Theorem 3.12 for such constants: $[\![\mathsf{K}]\!] : [\![\star]\!]\ \mathsf{K}\ \mathsf{K}$, but so are many other relations. Are we missing some restriction for constants? This question might be answered by resorting to a translation to pure terms via Church encodings (Böhm & Berarducci 1985) as Wadler (2007) does. However, in the hope to shed a different light on the issue, we give another explanation, using our machinery.

Consider a base type, such as $\mathsf{Bool} : \star$, equipped with constructors $\mathsf{true} : \mathsf{Bool}$ and $\mathsf{false} : \mathsf{Bool}$. In order to derive parametricity theorems in a system containing such a constant Bool, we must define $[\![\mathsf{Bool}]\!]$, satisfying $\vdash [\![\mathsf{Bool}]\!] : [\![\star]\!]\ \overline{\mathsf{Bool}}$. What are the restrictions put on the term $[\![\mathsf{Bool}]\!]$? First, we must be able to define $[\![\mathsf{true}]\!] : [\![\mathsf{Bool}]\!]\ \overline{\mathsf{true}}$. Therefore, $[\![\mathsf{Bool}]\!]\ \overline{\mathsf{true}}$ must be inhabited. The same reasoning holds for the false case.

Second, to write any useful program using Booleans, a way to test their value is needed. This may be done by adding a constant

$$\mathsf{if} : \mathsf{Bool} \to (\mathsf{A} : \star) \to \mathsf{A} \to \mathsf{A} \to \mathsf{A}$$

such that $\mathsf{if\ true\ A\ x\ y} \longrightarrow_\beta \mathsf{x}$ and $\mathsf{if\ false\ A\ x\ y} \longrightarrow_\beta \mathsf{y}$.

Now, if a program uses if, we must also define $[\![\mathsf{if}]\!]$ of type

$$[\![\mathsf{Bool} \to (\mathsf{A} : \star) \to \mathsf{A} \to \mathsf{A} \to \mathsf{A}]\!]\ \overline{\mathsf{if}}$$

for parametricity to work. Let us expand the type of $[\![\mathsf{if}]\!]$ and attempt to give a definition case by case:

$$\begin{aligned}
[\![\mathsf{if}]\!] : {} & \{\mathsf{b}_1\ \mathsf{b}_2 : \mathsf{Bool}\} \to (\mathsf{b}_\mathsf{R} : [\![\mathsf{Bool}]\!]\ \mathsf{b}_1\ \mathsf{b}_2) \to \\
& \{\mathsf{A}_1\ \mathsf{A}_2 : \star\} \to (\mathsf{A}_\mathsf{R} : [\![\star]\!]\ \mathsf{A}_1\ \mathsf{A}_2) \to \\
& \{\mathsf{x}_1 : \mathsf{A}_1\} \to \{\mathsf{x}_2 : \mathsf{A}_2\} \to (\mathsf{x}_\mathsf{R} : \mathsf{A}_\mathsf{R}\ \mathsf{x}_1\ \mathsf{x}_2) \to \\
& \{\mathsf{y}_1 : \mathsf{A}_1\} \to \{\mathsf{y}_2 : \mathsf{A}_2\} \to (\mathsf{y}_\mathsf{R} : \mathsf{A}_\mathsf{R}\ \mathsf{y}_1\ \mathsf{y}_2) \to \\
& \mathsf{A}_\mathsf{R}\ (\mathsf{if\ b}_1\ \mathsf{A}_1\ \mathsf{x}_1\ \mathsf{y}_1)\ (\mathsf{if\ b}_2\ \mathsf{A}_2\ \mathsf{x}_2\ \mathsf{y}_2)
\end{aligned}$$

$$\begin{aligned}
&[\![\mathsf{if}]\!]\ \{\mathsf{true}\}\ \ \{\mathsf{true}\}\ \ \mathsf{b}_\mathsf{R}\ _\ \mathsf{x}_\mathsf{R}\ \mathsf{y}_\mathsf{R} = \mathsf{x}_\mathsf{R} \\
&[\![\mathsf{if}]\!]\ \{\mathsf{true}\}\ \ \{\mathsf{false}\}\ \mathsf{b}_\mathsf{R}\ _\ \mathsf{x}_\mathsf{R}\ \mathsf{y}_\mathsf{R} = ?_{tf}
\end{aligned}$$

$$[\![\mathsf{if}]\!]\ \{\mathsf{false}\}\ \{\mathsf{true}\}\ \ \mathsf{b}_R\ \_\ \mathsf{x}_R\ \mathsf{y}_R\ =\ ?_{ft}$$
$$[\![\mathsf{if}]\!]\ \{\mathsf{false}\}\ \{\mathsf{false}\}\ \mathsf{b}_R\ \_\ \mathsf{x}_R\ \mathsf{y}_R\ =\ \mathsf{y}_R$$

(From this example onwards, we use a layout convention to ease the reading of translated types: each triple of arguments, corresponding to one argument in the original function, is written on its own line if space permits.)

In order to complete the above definition, we must provide a type-correct term for each question mark. For $?_{tf}$, this means that we must construct a term of type $\mathsf{A}_R\ \mathsf{x}_1\ \mathsf{y}_2$. Neither $\mathsf{x}_R : \mathsf{A}_R\ \mathsf{x}_1\ \mathsf{x}_2$ nor $\mathsf{y}_R : \mathsf{A}_R\ \mathsf{y}_1\ \mathsf{y}_2$ can help us here. The only liberty left is in $\mathsf{b}_R : [\![\mathsf{Bool}]\!]\ \mathsf{true}\ \mathsf{false}$. If we let $[\![\mathsf{Bool}]\!]\ \mathsf{true}\ \mathsf{false}$ be falsity ($\bot$, the empty type), then this case can never be reached and we need not give an equation for it. This reasoning holds symmetrically for $?_{ft}$. Therefore, we have the restrictions:

$$[\![\mathsf{Bool}]\!]\ \mathsf{x}\ \mathsf{x}\ = \text{some inhabited type}$$
$$[\![\mathsf{Bool}]\!]\ \mathsf{x}\ \mathsf{y}\ =\ \bot\quad \text{if } \mathsf{x} \neq \mathsf{y}$$

We have some freedom regarding picking "some inhabited type", so we choose $[\![\mathsf{Bool}]\!]\ \mathsf{x}\ \mathsf{x}$ to be truth ($\top$), making $[\![\mathsf{Bool}]\!]$ an encoding of the identity relation.

An intuition behind parametricity is that, when programs "know" more about a type, the parametricity condition becomes stronger. The above example illustrates how this intuition can be captured within our framework: having the eliminator if constrains the interpretation of Bool. We will make further use of this in Section 7.2.

5.2 Inductive families

Many languages permit datatype declarations for Bool, Nat, List, etc. Dependently typed languages typically allow the return types of constructors to have different arguments, yielding *inductive families* (Paulin-Mohring 1993; Dybjer 1994) such as the family Vec, in which the type is indexed by the number of elements. In Figure 3 we introduce Agda **data** syntax and some example datatypes and inductive families, which will be used later, including the sigma type, Σ which contains (dependent) pairs and the identity relation $\_\equiv\_$ which contains proofs of reflexivity. We sometimes write $(\mathsf{x} : A) \times B$ for $\Sigma\ A\ (\lambda \mathsf{x} : A.\ B)$, and elements of this type as (a, b), omitting the arguments A and $\lambda x : A.\ B$, handled by implicit syntax. For any values x and y of type A, the term $\mathsf{x} \equiv \mathsf{y}$ is a type, but only the types on the diagonal $\mathsf{x} \equiv \mathsf{x}$ are inhabited (by the canonical term refl).

In an "impure" PTS setting, datatype declarations can be interpreted as a simultaneous declaration of formation and introduction constants and also an eliminator and rules to analyse values of that datatype.

Example 5.1

The definition of List in Figure 3 gives rise to the following constants and rules:

$$\begin{array}{ll} \mathsf{List} & : (\mathsf{A} : \star) \to \star \\ \mathsf{nil} & : \{\mathsf{A} : \star\} \to \mathsf{List\ A} \\ \mathsf{cons} & : \{\mathsf{A} : \star\} \to \mathsf{A} \to \mathsf{List\ A} \to \mathsf{List\ A} \end{array}$$

```
data ⊥ : ⋆ where
  -- no constructors
data ⊤ : ⋆ where
  tt : ⊤
data Bool : ⋆ where
  false : Bool
  true  : Bool
data Nat : ⋆ where
  zero : Nat
  succ : Nat → Nat
```

```
data List (A : ⋆) : ⋆ where
  nil  : List A
  cons : A → List A → List A
data Vec (A : ⋆) : Nat → ⋆ where
  nilV  : Vec A zero
  consV : A → (n : Nat) → Vec A n → Vec A (succ n)
data Σ (A : ⋆) (B : A → ⋆) : ⋆ where
  _,_ : (a : A) → B a → Σ A B
data _≡_ {A : ⋆} (a : A) : A → ⋆ where
  refl : a ≡ a
```

Fig. 3. Examples of simple datatypes and inductive families (introducing Agda datatype syntax through well-known examples).

```
List-elim : {A : ⋆} → (P : List A → ⋆) →
            (base : P nil) →
            (step : (x : A) → (xs : List A) → P xs → P (cons x xs)) →
            (ys : List A) → P ys
List-elim P base step nil          = base
List-elim P base step (cons x xs)  = step x xs (List-elim P base step xs)
```

Note that the datatype parameter A is an implicit parameter of the constructor and eliminator constants.

More generally, family declarations of sort s (⋆ in the examples) have the typical form:[2]

$$\begin{array}{l}\textbf{data}\ \mathfrak{T}\ (\mathsf{a}:A) : (\mathsf{n}:N) \to s\ \textbf{where}\\ \quad \mathfrak{c} : (\mathsf{b}:B) \to (\mathsf{u}:((\mathsf{x}:X) \to \mathfrak{T}\ \mathsf{a}\ i)) \to \mathfrak{T}\ \mathsf{a}\ v\end{array}$$

Arguments of the type constructor $\mathfrak{T}$ may be either parameters a, which scope over the constructors and are repeated at each recursive use of $\mathfrak{T}$, or indices n, which may vary between uses. Data constructors c have non-recursive arguments b, whose types are otherwise unrestricted, and recursive arguments u with types of a constrained form ($\mathfrak{T}$ can not appear in X).

In PTS style we have the following formation and introduction constants:

$$\begin{array}{ll}\mathfrak{T} : (\mathsf{a}:A) \to (\mathsf{n}:N) \to s & \text{-- type}\\ \mathfrak{c} : \{\mathsf{a}:A\} \to (\mathsf{b}:B) \to ((\mathsf{x}:X) \to \mathfrak{T}\ \mathsf{a}\ i) \to \mathfrak{T}\ \mathsf{a}\ v & \text{-- constructor}\end{array}$$

and also a corresponding eliminator:

$$\begin{array}{l}\mathfrak{T}\text{-elim} : \{\mathsf{a}:A\} \to\\ \qquad (\mathsf{P}:((\mathsf{n}:N) \to \mathfrak{T}\ \mathsf{a}\ \mathsf{n} \to s)) \to\\ \qquad \mathsf{Case}_{\mathfrak{c}} \to (\mathsf{n}:N) \to (\mathsf{t}:\mathfrak{T}\ \mathsf{a}\ \mathsf{n}) \to \mathsf{P}\ \mathsf{n}\ \mathsf{t}\end{array}$$

where the type $\mathsf{Case}_{\mathfrak{c}}$ of the case for each constructor c is

$$(\mathsf{b}:B) \to (\mathsf{u}:((\mathsf{x}:X) \to \mathfrak{T}\ \mathsf{a}\ i)) \to ((\mathsf{x}:X) \to \mathsf{P}\ i\ (\mathsf{u}\ \mathsf{x})) \to \mathsf{P}\ v\ (\mathfrak{c}\ \{\mathsf{a}\}\ \mathsf{b}\ \mathsf{u})$$

[2] We show only one of the each element (parameter a, index n, constructor c, etc.) here. The generalisation to arbitrary numbers is straightforward but notationally cumbersome.

with one evaluation rule (β-reduction) for each constructor c:

$$\mathfrak{T}\text{-elim } \{\mathsf{a}\}\ \mathsf{P}\ \mathsf{e}\ v\ (\mathsf{c}\ \{\mathsf{a}\}\ \mathsf{b}\ \mathsf{u}) = \mathsf{e}\ \mathsf{b}\ \mathsf{u}\ (\lambda \mathsf{x} : X.\ \mathfrak{T}\text{-elim } \{\mathsf{a}\}\ \mathsf{P}\ \mathsf{e}\ i\ (\mathsf{u}\ \mathsf{x})) \tag{1}$$

As in the List example, the datatype parameter A is an implicit parameter of the constructor and eliminator constants.

We often use corresponding pattern matching definitions instead of these eliminators (Coquand 1992).

In the following sections, we consider two ways to "generically" define a proof term $[\![\mathsf{c}]\!] : [\![T]\!]\ \mathsf{c} \ldots \mathsf{c}$ for each constant $\mathsf{c} : T$ introduced by the data definition.

5.3 Deductive-style translation

In Section 5.1 we gave a definition of $[\![\mathsf{Bool}]\!]$ and $[\![\mathsf{if}]\!]$ for a simplified eliminator if. In this subsection we present similar deductive-style translations for several concrete examples, and then deal with the general case. We define each proof as a term (using pattern matching to simplify the presentation) built up from simpler building blocks (pairs and units). (In Section 5.4 the inductive-style translation, we instead translate datatypes to families; **data** to **data**.)

Lists From the definition of List in Figure 3, we have the constant $\mathsf{List} : \star \to \star$, so List is an example of a type constructor, and thus $[\![\mathsf{List}]\!]$ should be a relation transformer. As with $[\![\mathsf{Bool}]\!]$, lists are related only if their constructors match. Two nil lists are trivially related; as in the Bool case we use $\top$ for the nullary constructor. Two cons lists are related only if their components are related; the proof of that relationship is a pair of proofs for the components, represented as a product ($\times$):

$$\begin{array}{lll}
[\![\mathsf{List}]\!] : [\![\star \to \star]\!]\ \mathsf{List}\ \mathsf{List} & & \\
[\![\mathsf{List}]\!]\ \mathsf{A_R}\ \mathsf{nil} & \mathsf{nil} & = \top \\
[\![\mathsf{List}]\!]\ \mathsf{A_R}\ (\mathsf{cons}\ \mathsf{x}_1\ \mathsf{xs}_1)\ (\mathsf{cons}\ \mathsf{x}_2\ \mathsf{xs}_2) & & = \mathsf{A_R}\ \mathsf{x}_1\ \mathsf{x}_2\ \times\ [\![\mathsf{List}]\!]\ \mathsf{A_R}\ \mathsf{xs}_1\ \mathsf{xs}_2 \\
[\![\mathsf{List}]\!]\ \mathsf{A_R}\ _ & _ & = \bot
\end{array}$$

This is exactly the definition of Wadler (1989): Lists are related iff their lengths are equal and their elements are related point-wise. The translations of the constructors build the corresponding proofs:

$$\begin{array}{l}
[\![\mathsf{nil}]\!] : [\![(\mathsf{A} : \star) \to \mathsf{List\ A}]\!]\ \mathsf{nil}\ \mathsf{nil} \\
[\![\mathsf{nil}]\!]\ \mathsf{A_R} = \mathsf{tt} \\
[\![\mathsf{cons}]\!] : [\![(\mathsf{A} : \star) \to \mathsf{A} \to \mathsf{List\ A} \to \mathsf{List\ A}]\!]\ \mathsf{cons}\ \mathsf{cons} \\
[\![\mathsf{cons}]\!]\ \mathsf{A_R}\ \mathsf{x_R}\ \mathsf{xs_R} = (\mathsf{x_R}, \mathsf{xs_R})
\end{array}$$

List rearrangements The first example of a parametric type examined by Wadler (1989) is the type of list rearrangements: $\mathsf{R} = (\mathsf{A} : \star) \to \mathsf{List\ A} \to \mathsf{List\ A}$. Intuitively, functions of type R know nothing about the actual argument type A, and therefore they can only produce the output list by taking elements from the input list. Here we recover that result as an instance of Theorem 3.12.

Applying the translation to R yields:

$$\begin{aligned}
&[\![R]\!] : R \to R \to \star \\
&[\![R]\!]\ r_1\ r_2 = \{A_1\ A_2 : \star\} \to (A_R : [\![\star]\!]\ A_1\ A_2) \to \\
&\qquad \{xs_1 : \mathsf{List}\ A_1\} \to \{xs_2 : \mathsf{List}\ A_2\} \to (xs_R : [\![\mathsf{List}]\!]\ A_R\ xs_1\ xs_2) \to \\
&\qquad [\![\mathsf{List}]\!]\ A_R\ (r_1\ A_1\ xs_1)\ (r_2\ A_2\ xs_2)
\end{aligned}$$

In words: Two list rearrangements r_1 and r_2 are related iff for all types A_1 and A_2 with relation A_R, and for all lists xs_1 and xs_2 point-wise related by A_R, the resulting lists $r_1\ A_1\ xs_1$ and $r_2\ A_2\ xs_2$ are also point-wise related by A_R. By Theorem 3.12, $[\![R]\!]\ r\ r$ holds for any term r of type R. This means that applying r preserves (point-wise) any relation existing between input lists of equal length. By specialising A_R to a function ($A_R\ a_1\ a_2 = f\ a_1 \equiv a_2$), we obtain the following well-known result:

$$\begin{aligned}
&(A_1\ A_2 : \star) \to (f : A_1 \to A_2) \to (xs : \mathsf{List}\ A_1) \to \\
&\quad \mathsf{map}\ f\ (r\ A_1\ xs) \equiv r\ A_2\ (\mathsf{map}\ f\ xs)
\end{aligned}$$

(This form relies on the facts that [[List]] preserves identities and composes with map.)

Proof terms We have seen that applying [[_]] to a type yields a parametricity property for terms of that type, and by Theorem 3.12 we can also apply [[_]] to a term of that type to obtain a proof of the property. As an example, consider a rearrangement function odds that returns every second element from a list:

$$\begin{aligned}
&\mathsf{odds} : (A : \star) \to \mathsf{List}\ A \to \mathsf{List}\ A \\
&\mathsf{odds}\ A\ \mathsf{nil} = \mathsf{nil} \\
&\mathsf{odds}\ A\ (\mathsf{cons}\ x\ \mathsf{nil}) = \mathsf{cons}\ x\ \mathsf{nil} \\
&\mathsf{odds}\ A\ (\mathsf{cons}\ x\ (\mathsf{cons}\ _\ xs)) = \mathsf{cons}\ x\ (\mathsf{odds}\ A\ xs)
\end{aligned}$$

Any list rearrangement function must satisfy the parametricity condition [[R]] seen above, and [[odds]] is a proof that odds satisfies parametricity. Expanding it yields:

$$\begin{aligned}
&[\![\mathsf{odds}]\!] : [\![(A : \star) \to \mathsf{List}\ A \to \mathsf{List}\ A]\!]\ \mathsf{odds}\ \mathsf{odds} \\
&[\![\mathsf{odds}]\!]\ A_R\ \{\mathsf{nil}\}\ \{\mathsf{nil}\}\ _ = \mathsf{tt} \\
&[\![\mathsf{odds}]\!]\ A_R\ \{\mathsf{cons}\ x_1\ \mathsf{nil}\}\ \{\mathsf{cons}\ x_2\ \mathsf{nil}\}\ (x_R, _) = (x_R, \mathsf{tt}) \\
&[\![\mathsf{odds}]\!]\ A_R\ \{\mathsf{cons}\ x_1\ (\mathsf{cons}\ _\ xs_1)\}\ \{\mathsf{cons}\ x_2\ (\mathsf{cons}\ _\ xs_2)\}\ (x_R, (_, xs_R)) = \\
&\quad (x_R, [\![\mathsf{odds}]\!]\ A_R\ \{xs_1\}\ \{xs_2\}\ xs_R)
\end{aligned}$$

We see (by textual matching of the definitions) that [[odds]] performs essentially the same computation as odds, on two lists in parallel. However, instead of building a new list, it keeps track of the relations (in the R-subscripted variables). This behaviour stems from the last two cases in the definition of [[odds]]. Performing such a computation is enough to prove the parametricity condition.

Vectors The translations of the constants of Vec are simple extensions of those for List, with an additional requirement that sizes be related by the identity relation [[Nat]]:

$$
\begin{aligned}
&[\![\mathsf{Vec}]\!] : [\![(A : \star) \to \mathsf{Nat} \to \star]\!]\ \overline{\mathsf{Vec}}\\
&[\![\mathsf{Vec}]\!]\ A_R\ n_R\ \mathsf{nilV}\ \mathsf{nilV} = \top\\
&[\![\mathsf{Vec}]\!]\ A_R\ \{\mathsf{succ}\ n_1\}\ \{\mathsf{succ}\ n_2\}\ n_R\ (\mathsf{consV}\ n_1\ x_1\ xs_1)\ (\mathsf{consV}\ n_2\ x_2\ xs_2) =\\
&\quad A_R\ x_1\ x_2 \times (n_R : [\![\mathsf{Nat}]\!]\ n_1\ n_2) \times [\![\mathsf{Vec}]\!]\ A_R\ n_R\\
&[\![\mathsf{Vec}]\!]\ A_R\ n_R\ xs_1\ xs_2 = \bot\\
&[\![\mathsf{nilV}]\!] : [\![\{A : \star\} \to \mathsf{Vec}\ A\ \mathsf{zero}]\!]\ \overline{\mathsf{nilV}}\\
&[\![\mathsf{nilV}]\!]\ A_R = \mathsf{tt}\\
&[\![\mathsf{consV}]\!] : [\![\{A : \star\} \to A \to (n : \mathsf{Nat}) \to \mathsf{Vec}\ A\ n \to \mathsf{Vec}\ A\ (\mathsf{succ}\ n)]\!]\ \overline{\mathsf{consV}}\\
&[\![\mathsf{consV}]\!]\ A_R\ x_R\ n_R\ xs_R = (x_R, (n_R, xs_R))
\end{aligned}
$$

In the List example above we omitted the translation of the elimination constant List-elim. Here we shall handle the more complex Vec-elim, which has the type

$$
\begin{aligned}
&\mathsf{Vec\text{-}elim} : \{A : \star\} \to\\
&\qquad (P : (n : \mathsf{Nat}) \to \mathsf{Vec}\ n\ A \to \star) \to\\
&\qquad (en : P\ \mathsf{zero}\ (\mathsf{nilV}\ A)) \to\\
&\qquad (ec : (x : A) \to (n : \mathsf{Nat}) \to (xs : \mathsf{Vec}\ n\ A) \to\\
&\qquad\quad P\ n\ xs \to P\ (\mathsf{succ}\ n)\ (\mathsf{consV}\ x\ n\ xs)) \to\\
&\qquad (n : \mathsf{Nat}) \to (v : \mathsf{Vec}\ n\ A) \to P\ n\ v
\end{aligned}
$$

The translation of this constant has a large type, but a simple definition:

$$
\begin{aligned}
&[\![\mathsf{Vec\text{-}elim}]\!] : [\![\ \{A : \star\} \to\\
&\qquad (P : (n : \mathsf{Nat}) \to \mathsf{Vec}\ n\ A \to \star) \to\\
&\qquad (en : P\ \mathsf{zero}\ (\mathsf{nilV}\ A)) \to\\
&\qquad (ec : (x : A) \to (n : \mathsf{Nat}) \to (xs : \mathsf{Vec}\ n\ A) \to\\
&\qquad\quad P\ n\ xs \to P\ (\mathsf{succ}\ n)\ (\mathsf{consV}\ x\ n\ xs)) \to\\
&\qquad (n : \mathsf{Nat}) \to (v : \mathsf{Vec}\ n\ A) \to P\ n\ v]\!]\ \overline{\mathsf{Vec\text{-}elim}}\\
&[\![\mathsf{Vec\text{-}elim}]\!]\ A_R\ P_R\ en_R\ ec_R\ \_\ \{\mathsf{nilV}\}\ \{\mathsf{nilV}\}\ \_ = en_R\\
&[\![\mathsf{Vec\text{-}elim}]\!]\ A_R\ P_R\ en_R\ ec_R\ n_R\ \{\mathsf{consV}\ x_1\ n_1\ xs_1\}\ \{\mathsf{consV}\ x_2\ n_2\ xs_2\}\ (x_R, (n_R, xs_R))\\
&\qquad = ec_R\ x_R\ n_R\ xs_R\ ([\![\mathsf{Vec\text{-}elim}]\!]\ A_R\ P_R\ en_R\ ec_R\ n_R\ xs_R)
\end{aligned}
$$

Dependent pairs Two pairs (a_1, b_1) and (a_2, b_2) are related by $[\![A \times B]\!]$ if their respective components are related (by $[\![A]\!]$ and $[\![B]\!]$). A constructive reading is that a proof that two pairs are related can be represented as a pair of proofs. This generalises nicely to the dependent case: a dependent pair (of the Σ type from Figure 3) translates to another dependent pair. That is, a pair $(a, b) : \Sigma\ A\ B$ (where $a : A$ and $b : B\ a$) translates to

$$([\![a]\!], [\![b]\!]) : [\![\Sigma]\!]\ [\![A]\!]\ [\![B]\!]\ (a_1, b_1)\ (a_2, b_2)$$

where

$$
\begin{aligned}
&[\![\Sigma]\!] : \{A_1\ A_2 : \star\}\ (A_R : [\![\star]\!]\ A_1\ A_2)\\
&\qquad \{B_1 : A_1 \to \star\}\ \{B_2 : A_2 \to \star\}\\
&\qquad\quad (B_R : \{a_1 : A_1\}\ \{a_2 : A_2\} \to A_R\ a_1\ a_2 \to [\![\star]\!]\ (B_1\ a_1)\ (B_2\ a_2)) \to\\
&\qquad [\![\star]\!]\ (\Sigma\ A_1\ B_1)\ (\Sigma\ A_2\ B_2)\\
&[\![\Sigma]\!]\ A_R\ B_R\ (a_1, b_1)\ (a_2, b_2) = \Sigma\ (A_R\ a_1\ a_2)\ (\lambda\ a_R \to B_R\ a_R\ b_1\ b_2)
\end{aligned}
$$

Inductive families – general case For the "typical form" of an inductive family we begin with the translation of Equation (1) for each constructor c:

$$[\![\mathfrak{T}\text{-elim } \{\mathsf{a}\}\ \mathsf{P}\ \mathsf{e}\ v]\!]\ \overline{(\mathsf{c}\ \{\mathsf{a}\}\ \mathsf{b}\ \mathsf{u})}\ ([\![\mathsf{c}]\!]\ \{\overline{\mathsf{a}}\}\ \mathsf{a}_\mathsf{R}\ \{\overline{\mathsf{b}}\}\ \mathsf{b}_\mathsf{R}\ \{\overline{\mathsf{u}}\}\ \mathsf{u}_\mathsf{R}) = [\![RHS]\!] \tag{2}$$

for $RHS = \mathsf{e}\ \mathsf{b}\ \mathsf{u}\ (\lambda \mathsf{x} : X.\ \mathfrak{T}\text{-elim } \{\mathsf{a}\}\ \mathsf{P}\ \mathsf{e}\ i\ (\mathsf{u}\ \mathsf{x}))$. To turn this into a pattern matching definition of $[\![\mathfrak{T}\text{-elim}]\!]$, we need a suitable definition of $[\![\mathsf{c}]\!]$, and similarly for the constructors in v. The only arguments of $[\![\mathsf{c}]\!]$ not already in scope are b_R and u_R, so we package them as a dependent pair because the type of u_R may depend on that of b_R. We define

$$\begin{aligned}
&[\![\mathfrak{T}]\!] : [\![(\mathsf{a} : A) \to (\mathsf{n} : N) \to s]\!]\ \overline{\mathfrak{T}} \\
&[\![\mathfrak{T}]\!]\ \{\overline{\mathsf{a}}\}\ \mathsf{a}_\mathsf{R}\ \{\overline{v}\}\ [\![v]\!]\ \overline{(\mathsf{c}\ \{\mathsf{a}\}\ \mathsf{b}\ \mathsf{u})} = (\mathsf{b}_\mathsf{R} : [\![B]\!]\ \overline{\mathsf{b}}) \times [\![(\mathsf{x} : X) \to \mathfrak{T}\ \mathsf{a}\ i]\!]\ \overline{\mathsf{u}} \\
&[\![\mathfrak{T}]\!]\ \{\overline{\mathsf{a}}\}\ \mathsf{a}_\mathsf{R}\ \{\overline{\mathsf{u}}\}\ \mathsf{u}_\mathsf{R}\ \overline{\mathsf{t}} \qquad\qquad = \bot \\
&[\![\mathsf{c}]\!] : [\![(\{\mathsf{a} : A\}) \to (\mathsf{b} : B) \to ((\mathsf{x} : X) \to \mathfrak{T}\ \mathsf{a}\ i) \to \mathfrak{T}\ \mathsf{a}\ v]\!]\ \overline{\mathsf{c}} \\
&[\![\mathsf{c}]\!]\ \mathsf{a}_\mathsf{R}\ \mathsf{b}_\mathsf{R}\ \mathsf{u}_\mathsf{R} = (\mathsf{b}_\mathsf{R}, \mathsf{u}_\mathsf{R})
\end{aligned}$$

Substituting the above definition of $[\![\mathsf{c}]\!]$ into Equation (2), we obtain a clause for the definition of $[\![\mathfrak{T}\text{-elim}]\!]$:

$$[\![\mathfrak{T}\text{-elim } \{\mathsf{a}\}\ \mathsf{P}\ \mathsf{e}\ v]\!]\ \overline{(\mathsf{c}\ \{\mathsf{a}\}\ \mathsf{b}\ \mathsf{u})}\ (\mathsf{b}_\mathsf{R}, \mathsf{u}_\mathsf{R}) = [\![RHS]\!]$$

These clauses cover only cases where the constructors match, but because $[\![\mathfrak{T}]\!]$ yields $\bot$ otherwise, that is complete coverage.

The question whether the translation of the eliminator and its reduction rule are inductively well-founded is delayed until we have completed the presentation of the Inductive-style translation.

5.4 Inductive-style translation

Another way of defining the translations $[\![\mathsf{c}]\!]$ of the constants associated with a datatype is to use an *inductive* definition (using **data**) in contrast with the *deductive* definitions (construction using pairs and units) of the previous section.

Deductive- and inductive-style translations define the same relation, but the objects witnessing the instances of the inductively defined relation record additional information, namely which rules are used to prove membership of the relation. However, since the same constructor never appears in more than one case of the inductive definition, that additional content can be recovered from a witness of the deductive-style definition; therefore, the two styles are isomorphic. This will become clear in the upcoming examples.

Booleans For the **data**-declaration of Bool (from Figure 3), we can define translations of the datatype and its constructors directly with another inductive definition:

$$\begin{aligned}
&\textbf{data}\ [\![\mathsf{Bool}]\!] : [\![\star]\!]\ \overline{\mathsf{Bool}}\ \textbf{where} \\
&\quad [\![\mathsf{true}]\!] : [\![\mathsf{Bool}]\!]\ \overline{\mathsf{true}} \\
&\quad [\![\mathsf{false}]\!] : [\![\mathsf{Bool}]\!]\ \overline{\mathsf{false}}
\end{aligned}$$

The main difference from the deductive-style definition is that it is possible, by analysis of a value of type $[\![\mathsf{Bool}]\!]$, to recover the arguments of the relation (either all true, or all false).

The elimination constant for Bool is

$$\mathsf{Bool\text{-}elim} : (\mathsf{P} : \mathsf{Bool} \to \star) \to \mathsf{P\ true} \to \mathsf{P\ false} \to (\mathsf{b} : \mathsf{Bool}) \to \mathsf{P\ b}$$

Similarly, our new datatype $[\![\mathsf{Bool}]\!]$ (with arity $n = 2$) has an elimination constant with the following type:

$$\begin{aligned}[\![\mathsf{Bool}]\!]\text{-}\mathsf{elim} : {} & (\mathsf{C} : (\mathsf{a}_1\ \mathsf{a}_2 : \mathsf{Bool}) \to [\![\mathsf{Bool}]\!]\ \mathsf{a}_1\ \mathsf{a}_2 \to \star) \to \\ & \mathsf{C\ true\ true}\ [\![\mathsf{true}]\!] \to \mathsf{C\ false\ false}\ [\![\mathsf{false}]\!] \to \\ & \{\mathsf{b}_1\ \mathsf{b}_2 : \mathsf{Bool}\} \to (\mathsf{b}_R : [\![\mathsf{Bool}]\!]\ \mathsf{b}_1\ \mathsf{b}_2) \to \mathsf{C}\ \mathsf{b}_1\ \mathsf{b}_2\ \mathsf{b}_R\end{aligned}$$

We can define $[\![\mathsf{Bool\text{-}elim}]\!]$ using the elimination constants Bool-elim and $[\![\mathsf{Bool}]\!]$-elim as follows:

$$\begin{aligned}&[\![\mathsf{Bool\text{-}elim}]\!] : \\ &\quad \{\mathsf{P}_1\ \mathsf{P}_2 : \mathsf{Bool} \to \star\} \to (\mathsf{P}_R : [\![\mathsf{Bool} \to \star]\!]\ \mathsf{P}_1\ \mathsf{P}_2) \to \\ &\quad \{\mathsf{x}_1 : \mathsf{P}_1\ \mathsf{true}\} \to \{\mathsf{x}_2 : \mathsf{P}_2\ \mathsf{true}\} \to (\mathsf{P}_R\ [\![\mathsf{true}]\!]\ \mathsf{x}_1\ \mathsf{x}_2) \to \\ &\quad \{\mathsf{y}_1 : \mathsf{P}_1\ \mathsf{false}\} \to \{\mathsf{y}_2 : \mathsf{P}_2\ \mathsf{false}\} \to (\mathsf{P}_R\ [\![\mathsf{false}]\!]\ \mathsf{y}_1\ \mathsf{y}_2) \to \\ &\quad \{\mathsf{b}_1\ \mathsf{b}_2 : \mathsf{Bool}\} \to (\mathsf{b}_R : [\![\mathsf{Bool}]\!]\ \mathsf{b}_1\ \mathsf{b}_2) \to \\ &\quad \mathsf{P}_R\ \mathsf{b}_R\ (\mathsf{Bool\text{-}elim}\ \mathsf{P}_1\ \mathsf{x}_1\ \mathsf{y}_1\ \mathsf{b}_1) \\ &\qquad\quad (\mathsf{Bool\text{-}elim}\ \mathsf{P}_2\ \mathsf{x}_2\ \mathsf{y}_2\ \mathsf{b}_2) \\ &[\![\mathsf{Bool\text{-}elim}]\!]\ \{\mathsf{P}_1\}\ \{\mathsf{P}_2\}\ \mathsf{P}_R\ \{\mathsf{x}_1\}\ \{\mathsf{x}_2\}\ \mathsf{x}_R\ \{\mathsf{y}_1\}\ \{\mathsf{y}_2\}\ \mathsf{y}_R \\ &\quad = [\![\mathsf{Bool}]\!]\text{-}\mathsf{elim} \\ &\qquad\quad (\lambda\ \mathsf{b}_1\ \mathsf{b}_2\ \mathsf{b}_R \to \mathsf{P}_R\ \mathsf{b}_R\ (\mathsf{Bool\text{-}elim}\ \mathsf{P}_1\ \mathsf{x}_1\ \mathsf{y}_1\ \mathsf{b}_1) \\ &\qquad\qquad\qquad\qquad\quad (\mathsf{Bool\text{-}elim}\ \mathsf{P}_2\ \mathsf{x}_2\ \mathsf{y}_2\ \mathsf{b}_2)) \\ &\qquad\quad \mathsf{x}_R\ \mathsf{y}_R\end{aligned}$$

Lists For List, as introduced in Figure 3, we can again define translations of the datatype and its constructors with a corresponding new inductive definition:

$$\begin{aligned}&\mathbf{data}\ [\![\mathsf{List}]\!]\ ([\![\mathsf{A} : \star]\!]) : [\![\star]\!]\ \overline{(\mathsf{List\ A})}\ \mathbf{where} \\ &\quad [\![\mathsf{nil}]\!] \quad : [\![\mathsf{List\ A}]\!]\ \overline{\mathsf{nil}} \\ &\quad [\![\mathsf{cons}]\!] : [\![\mathsf{A} \to \mathsf{List\ A} \to \mathsf{List\ A}]\!]\ \overline{\mathsf{cons}}\end{aligned}$$

or after expansion (for $n = 2$):

$$\begin{aligned}&\mathbf{data}\ [\![\mathsf{List}]\!]\ \{\mathsf{A}_1\ \mathsf{A}_2 : \star\}\ (\mathsf{A}_R : [\![\star]\!]\ \mathsf{A}_1\ \mathsf{A}_2) : \mathsf{List\ A}_1 \to \mathsf{List\ A}_2 \to \star\ \mathbf{where} \\ &\quad [\![\mathsf{nil}]\!] \quad : [\![\mathsf{List}]\!]\ \mathsf{A}_R\ \mathsf{nil}\ \mathsf{nil} \\ &\quad [\![\mathsf{cons}]\!] : \{\mathsf{x}_1 : \mathsf{A}_1\} \to \{\mathsf{x}_2 : \mathsf{A}_2\} \to (\mathsf{x}_R : \mathsf{A}_R\ \mathsf{x}_1\ \mathsf{x}_2) \to \\ &\qquad\quad \{\mathsf{xs}_1 : \mathsf{List\ A}_1\} \to \{\mathsf{xs}_2 : \mathsf{List\ A}_2\} \to (\mathsf{xs}_R : [\![\mathsf{List}]\!]\ \mathsf{A}_R\ \mathsf{xs}_1\ \mathsf{xs}_2) \to \\ &\qquad\quad [\![\mathsf{List}]\!]\ \mathsf{A}_R\ (\mathsf{cons}\ \mathsf{x}_1\ \mathsf{xs}_1) \\ &\qquad\qquad\qquad\quad (\mathsf{cons}\ \mathsf{x}_2\ \mathsf{xs}_2)\end{aligned}$$

The above definition encodes the same relational action as that given in Section 5.3. Again, the difference is that the *derivation* of a relation between lists xs_1 and xs_2 is available as an object of type $[\![\mathsf{List}]\!]\ \mathsf{A}_R\ \mathsf{xs}_1\ \mathsf{xs}_2$.

Proof terms The proof term for the list-rearrangement example can be constructed in a similar way to the deductive one. The main difference is that the target lists are also built and recorded in the ⟦List⟧ structure. In short, this version has more of a computational flavour than the deductive version,

$$
\begin{aligned}
&⟦\mathsf{odds}⟧ : ⟦(A : \star) \to \mathsf{List}\ A \to \mathsf{List}\ A⟧\ \mathsf{odds}\ \mathsf{odds}\\
&⟦\mathsf{odds}⟧\ A_R\ ⟦\mathsf{nil}⟧ &&= ⟦\mathsf{nil}⟧\quad A_R\\
&⟦\mathsf{odds}⟧\ A_R\ (⟦\mathsf{cons}⟧\ x_R\ ⟦\mathsf{nil}⟧) &&= ⟦\mathsf{cons}⟧\ A_R\ x_R\ (⟦\mathsf{nil}⟧\quad A_R)\\
&⟦\mathsf{odds}⟧\ A_R\ (⟦\mathsf{cons}⟧\ x_R\ (⟦\mathsf{cons}⟧\ \_\ xs_R)) &&= ⟦\mathsf{cons}⟧\ A_R\ x_R\ (⟦\mathsf{odds}⟧\ A_R\ xs_R)
\end{aligned}
$$

Vectors We can apply the same translation method to inductive families. For example, the translation of the family Vec of lists indexed by their length is

$$
\begin{aligned}
&\textbf{data}\ ⟦\mathsf{Vec}⟧\ (⟦A : \star⟧) : ⟦\mathsf{Nat} \to \star⟧\ \overline{(\mathsf{Vec}\ A)}\ \textbf{where}\\
&\quad ⟦\mathsf{nilV}⟧\ : ⟦\mathsf{Vec}\ A\ \mathsf{zero}⟧\ \overline{\mathsf{nilV}}\\
&\quad ⟦\mathsf{consV}⟧\ : ⟦\{x : A\} \to (n : \mathsf{Nat}) \to \mathsf{Vec}\ A\ n \to \mathsf{Vec}\ A\ (\mathsf{succ}\ n)⟧\ \overline{\mathsf{consV}}
\end{aligned}
$$

or, if we expand the translation of the types:

$$
\begin{aligned}
&\textbf{data}\ ⟦\mathsf{Vec}⟧\ \{A_1\ A_2 : \star\}\ (A_R : A_1 \to A_2 \to \star) :\\
&\qquad \{n_1\ n_2 : \mathsf{Nat}\} \to (n_R : ⟦\mathsf{Nat}⟧\ n_1\ n_2) \to\\
&\qquad \mathsf{Vec}\ A_1\ n_1 \to \mathsf{Vec}\ A_2\ n_2 \to \star\ \textbf{where}\\
&\quad ⟦\mathsf{nilV}⟧\ : ⟦\mathsf{Vec}⟧\ A_R\ ⟦\mathsf{zero}⟧\ \mathsf{nilV}\ \mathsf{nilV}\\
&\quad ⟦\mathsf{consV}⟧\ : \{x_1 : A_1\} \to \{x_2 : A_2\} \to (x_R : A_R\ x_1\ x_2) \to\\
&\qquad\quad \{n_1\ n_2 : \mathsf{Nat}\} \to (n_R : ⟦\mathsf{Nat}⟧\ n_1\ n_2) \to\\
&\qquad\quad \{xs_1 : \mathsf{Vec}\ A_1\ n_1\} \to \{xs_2 : \mathsf{Vec}\ A_2\ n_2\} \to\\
&\qquad\qquad (xs_R : ⟦\mathsf{Vec}⟧\ A_R\ n_R\ xs_1\ xs_2) \to\\
&\qquad\quad ⟦\mathsf{Vec}⟧\ A_R\ (⟦\mathsf{succ}⟧\ n_R)\ (\mathsf{consV}\ x_1\ n_1\ xs_1)\ (\mathsf{consV}\ x_2\ n_2\ xs_2)
\end{aligned}
$$

The relation obtained by applying ⟦_⟧ encodes that vectors are related if their lengths are the same and their elements are related point-wise. The difference with the List version is that the equality of lengths is encoded in ⟦consV⟧ as an ⟦Nat⟧ (identity) relation.

As in the Bool case, we can define the translation of Vec-elim in terms of ⟦Vec⟧-elim:

$$
\begin{aligned}
&⟦\ \mathsf{Vec\text{-}elim}\ ⟧ : ⟦\ \{A : \star\} \to\\
&\qquad\qquad (P : (n : \mathsf{Nat}) \to \mathsf{Vec}\ n\ A \to \star) \to\\
&\qquad\qquad (\mathsf{en} : P\ \mathsf{zero}\ (\mathsf{nilV}\ A)) \to\\
&\qquad\qquad (\mathsf{ec} : (x : A) \to (n : \mathsf{Nat}) \to (xs : \mathsf{Vec}\ n\ A) \to\\
&\qquad\qquad\qquad P\ n\ xs \to P\ (\mathsf{succ}\ n)\ (\mathsf{consV}\ A\ x\ n\ xs)) \to\\
&\qquad\qquad (n : \mathsf{Nat}) \to (v : \mathsf{Vec}\ n\ A) \to P\ n\ v⟧\ \overline{\mathsf{Vec\text{-}elim}}\\
&⟦\ \mathsf{Vec\text{-}elim}\ A\ P\ \mathsf{en}\ \mathsf{ec}⟧\ =\ ⟦\mathsf{Vec}⟧\text{-}\mathsf{elim}\ A_R\\
&\quad (\lambda\ ⟦n : \mathsf{Nat}, v : \mathsf{Vec}\ n\ A⟧\ .\ ⟦P\ n\ v⟧\ \overline{(\mathsf{Vec\text{-}elim}\ A\ P\ \mathsf{en}\ \mathsf{ec}\ v)})\\
&\quad \mathsf{en}_R\\
&\quad (\lambda\ ⟦x : A, n : \mathsf{Nat}, xs : \mathsf{Vec}\ n\ A⟧\ .\ ⟦\mathsf{ec}\ x\ n\ xs⟧\ \overline{(\mathsf{Vec\text{-}elim}\ A\ P\ \mathsf{en}\ \mathsf{ec}\ xs)})
\end{aligned}
$$

Inductive families – general case Starting from an inductive family of the same typical form as in the previous section,

$$\begin{array}{l}\textbf{data}\ \mathfrak{T}\ (\mathsf{a}:A):K\ \textbf{where}\\ \quad \mathfrak{c}:C\end{array}$$

where $K = (\mathsf{n}:N) \to s$ and $C = (\mathsf{b}:B) \to ((\mathsf{x}:X) \to \mathfrak{T}\ \mathsf{a}\ i) \to \mathfrak{T}\ \mathsf{a}\ v$, by applying our translation to the components of the **data**-declaration, we obtain an inductive family that defines the relational counterparts of the original type $\mathfrak{T}$ and its constructors $\mathfrak{c}$ at the same time:

$$\begin{array}{l}\textbf{data}\ [\![\mathfrak{T}]\!]\ [\![\mathsf{a}:A]\!] : [\![K]\!]\ \overline{(\mathfrak{T}\ \mathsf{a})}\ \textbf{where}\\ \quad [\![\mathfrak{c}]\!] : [\![C]\!]\ \overline{(\mathfrak{c}\ \{\mathsf{a}\})}\end{array}$$

It remains to supply a proof term for the parametricity of the elimination constant $\mathfrak{T}$-elim. We start by inlining C and K; the inductive family is parametrised on A, indexed by N, and has the form

$$\begin{array}{l}\textbf{data}\ \mathfrak{T}\ (\mathsf{a}:A) : (\mathsf{n}:N) \to s\ \textbf{where}\\ \quad \mathfrak{c} : (\mathsf{b}:B) \to ((\mathsf{x}:X) \to \mathfrak{T}\ \mathsf{a}\ i) \to \mathfrak{T}\ \mathsf{a}\ v\end{array}$$

The translated family is parametrised by a relation on $\overline{A}$ and lifts relations on $\overline{N}$ to relations on $\mathfrak{T}$ a n. The definition follows from mechanical application of $[\![\_]\!]$ to K and C:

$$\begin{array}{l}\textbf{data}\ [\![\mathfrak{T}]\!]\ \overline{(\mathsf{a}:A)}\ (\mathsf{a}_\mathsf{R}:[\![A]\!]\ \overline{\mathsf{a}})\ :\ \{\overline{\mathsf{n}:N}\} \to (\mathsf{n}_\mathsf{R}:[\![N]\!]\ \overline{\mathsf{n}}) \to \overline{\mathfrak{T}\ \mathsf{a}\ \mathsf{n}} \to \widetilde{s}\ \ \textbf{where}\\ \quad [\![\mathfrak{c}]\!] : \{\overline{\mathsf{b}:B}\} \to (\mathsf{b}_\mathsf{R}:[\![B]\!]\ \overline{\mathsf{b}}) \to [\![((\mathsf{x}:X) \to \mathfrak{T}\ \mathsf{a}\ i) \to \mathfrak{T}\ \mathsf{a}\ v]\!]\ \overline{(\mathfrak{c}\ \{\mathsf{a}\}\ b)}\end{array}$$

Each inductive family comes with an elimination constant, and for elimination of $[\![\mathfrak{T}]\!]$ to sort $\widetilde{s_e}$ it has type

$$\begin{array}{l}[\![\mathfrak{T}]\!]\text{-elim} : \{\overline{\mathsf{a}:A}\} \to \{\mathsf{a}_\mathsf{R}:[\![A]\!]\ \overline{\mathsf{a}}\} \to\\ \qquad (\mathsf{Q}:\{\overline{\mathsf{n}:N}\} \to (\mathsf{n}_\mathsf{R}:[\![N]\!]\ \overline{\mathsf{n}}) \to (\overline{\mathsf{t}:\mathfrak{T}\ \mathsf{a}\ \mathsf{n}}) \to [\![\mathfrak{T}\ \mathsf{a}\ \mathsf{n}]\!]\ \bar{\mathsf{t}} \to \widetilde{s_e}) \to\\ \qquad \mathsf{Case}_{[\![\mathfrak{c}]\!]} \to\\ \qquad \{\overline{\mathsf{n}:N}\} \to (\mathsf{n}_\mathsf{R}:[\![N]\!]\ \overline{\mathsf{n}}) \to (\overline{\mathsf{t}:\mathfrak{T}\ \mathsf{a}\ \mathsf{n}}) \to (\mathsf{t}_\mathsf{R}:[\![\mathfrak{T}\ \mathsf{a}\ \mathsf{n}]\!]\ \bar{\mathsf{t}}) \to \mathsf{Q}\ \{\overline{\mathsf{n}}\}\ \mathsf{n}_\mathsf{R}\ \bar{\mathsf{t}}\ \mathsf{t}_\mathsf{R}\end{array}$$

where $\mathsf{Case}_{[\![\mathfrak{c}]\!]}$ is

$$\begin{array}{l}\{\overline{\mathsf{b}:B}\} \to (\mathsf{b}_\mathsf{R}:[\![B]\!]\ \overline{\mathsf{b}}) \to\\ \{\overline{\mathsf{u}:(\mathsf{x}:X) \to \mathfrak{T}\ \mathsf{a}\ i}\} \to (\mathsf{u}_\mathsf{R}:[\![(\mathsf{x}:X) \to \mathfrak{T}\ \mathsf{a}\ i]\!]\ \overline{\mathsf{u}}) \to\\ (\{\overline{\mathsf{x}:X}\} \to (\mathsf{x}_\mathsf{R}:[\![X]\!]\ \overline{\mathsf{x}}) \to \mathsf{Q}\ \{\bar{i}\}\ [\![i]\!]\ \overline{(\mathsf{u}\ \mathsf{x})}\ [\![\mathsf{u}\ \mathsf{x}]\!]) \to\\ \mathsf{Q}\ \{\bar{v}\}\ [\![v]\!]\ \overline{(\mathfrak{c}\ \{\mathsf{a}\}\ \mathsf{b}\ \mathsf{u})}\ [\![\mathfrak{c}\ \{\mathsf{a}\}\ \mathsf{b}\ \mathsf{u}]\!]\end{array}$$

Using the eliminator ($[\![\mathfrak{T}]\!]$-elim) of the translated family and the eliminator ($\mathfrak{T}$-elim) of the original family, the proof term $[\![\mathfrak{T}\text{-elim}]\!]$ can be defined as follows:

$$\begin{array}{l}[\![\mathfrak{T}\text{-elim}]\!] : [\![\{\mathsf{a}:A\} \to (\mathsf{P}:((\mathsf{n}:N) \to \mathfrak{T}\ \mathsf{a}\ \mathsf{n} \to s)) \to (\mathsf{e}:\mathsf{Case}_\mathfrak{c}) \to\\ \qquad\qquad (\mathsf{n}:N) \to (\mathsf{t}:\mathfrak{T}\ \mathsf{a}\ \mathsf{n}) \to \mathsf{P}\ \mathsf{n}\ \mathsf{t}]\!]\ \overline{\mathfrak{T}\text{-elim}}\\ [\![\mathfrak{T}\text{-elim}\ \{\mathsf{a}\}\ \mathsf{P}\ \mathsf{e}]\!] = [\![\mathfrak{T}]\!]\text{-elim}\ \{\overline{\mathsf{a}}\}\ \{\mathsf{a}_\mathsf{R}\}\ \mathsf{Q}\ \mathsf{f}\end{array}$$

where

$$\mathsf{Q}\ \{\overline{\mathsf{n}}\}\ \mathsf{n}_\mathsf{R}\ \bar{\mathsf{t}}\ \mathsf{t}_\mathsf{R} = [\![\mathsf{P}\ \mathsf{n}\ \mathsf{t}]\!]\ \overline{(\mathfrak{T}\text{-elim}\ \{\mathsf{a}\}\ \mathsf{P}\ \mathsf{e}\ \mathsf{n}\ \mathsf{t})} \tag{3}$$

$$\mathsf{f}\ \{\overline{\mathsf{b}}\}\ \mathsf{b}_\mathsf{R}\ \{\overline{\mathsf{u}}\}\ \mathsf{u}_\mathsf{R} = [\![\mathsf{e}\ \mathsf{b}\ \mathsf{u}]\!]\ \{\overline{(\lambda \mathsf{x} : X.\ \mathfrak{T}\text{-elim}\ \{\mathsf{a}\}\ \mathsf{P}\ \mathsf{e}\ i\ (\mathsf{u}\ \mathsf{x}))}\} \tag{4}$$

We proceed to check that f has the right return type. Because

$$\mathsf{e}\ \mathsf{b}\ \mathsf{u} : ((\mathsf{x} : X) \to \mathsf{P}\ i\ (\mathsf{u}\ \mathsf{x})) \to \mathsf{P}\ v\ (\mathsf{c}\ \{\mathsf{a}\}\ \mathsf{b}\ \mathsf{u})$$

we have (by the abstraction theorem)

$$\begin{aligned}[\![\mathsf{e}\ \mathsf{b}\ \mathsf{u}]\!] : \{&\overline{\mathsf{p} : (\mathsf{x} : X) \to \mathsf{P}\ i\ (\mathsf{u}\ \mathsf{x})}\} \to \\ &(\{\overline{\mathsf{x} : X}\} \to (\mathsf{x}_\mathsf{R} : [\![X]\!]\ \overline{\mathsf{x}}) \to [\![\mathsf{P}\ i\ (\mathsf{u}\ \mathsf{x})]\!]\ \overline{(\mathsf{p}\ \mathsf{x})}) \to \\ &[\![\mathsf{P}\ v\ (\mathsf{c}\ \{\mathsf{a}\}\ \mathsf{b}\ \mathsf{u})]\!]\ \overline{(\mathsf{e}\ \mathsf{b}\ \mathsf{u}\ \mathsf{p})}\end{aligned}$$

and hence the type of $\mathsf{f}\ \{\overline{\mathsf{b}}\}\ \mathsf{b}_\mathsf{R}\ \{\overline{\mathsf{u}}\}\ \mathsf{u}_\mathsf{R}$ is:

$$\begin{aligned}&(\{\overline{\mathsf{x} : X}\} \to (\mathsf{x}_\mathsf{R} : [\![X]\!]\ \overline{\mathsf{x}}) \to [\![\mathsf{P}\ i\ (\mathsf{u}\ \mathsf{x})]\!]\ \overline{(\mathfrak{T}\text{-elim}\ \{\mathsf{a}\}\ \mathsf{P}\ \mathsf{e}\ i\ (\mathsf{u}\ \mathsf{x}))}) \to \\ &[\![\mathsf{P}\ v\ (\mathsf{c}\ \{\mathsf{a}\}\ \mathsf{b}\ \mathsf{u})]\!]\ \overline{(\mathsf{e}\ \mathsf{b}\ \mathsf{u}\ (\lambda \mathsf{x} : X.\ \mathfrak{T}\text{-elim}\ \{\mathsf{a}\}\ \mathsf{P}\ \mathsf{e}\ i\ (\mathsf{u}\ \mathsf{x})))} \\ =\ &\{\ \text{datatype equation (1) from page 19}\ \} \\ &(\{\overline{\mathsf{x} : X}\} \to (\mathsf{x}_\mathsf{R} : [\![X]\!]\ \overline{\mathsf{x}}) \to [\![\mathsf{P}\ i\ (\mathsf{u}\ \mathsf{x})]\!]\ \overline{(\mathfrak{T}\text{-elim}\ \{\mathsf{a}\}\ \mathsf{P}\ \mathsf{e}\ i\ (\mathsf{u}\ \mathsf{x}))}) \to \\ &[\![\mathsf{P}\ v\ (\mathsf{c}\ \{\mathsf{a}\}\ \mathsf{b}\ \mathsf{u})]\!]\ \overline{(\mathfrak{T}\text{-elim}\ \{\mathsf{a}\}\ \mathsf{P}\ \mathsf{e}\ v\ (\mathsf{c}\ \{\mathsf{a}\}\ \mathsf{b}\ \mathsf{u}))} \\ =\ &\{\ \text{definition of Q (3)}\ \} \\ &(\{\overline{\mathsf{x} : X}\} \to (\mathsf{x}_\mathsf{R} : [\![X]\!]\ \overline{\mathsf{x}}) \to \mathsf{Q}\ \{\bar{i}\}\ [\![i]\!]\ (\overline{\mathsf{u}\ \mathsf{x}})\ [\![\mathsf{u}\ \mathsf{x}]\!]) \to \\ &\mathsf{Q}\ \{\bar{v}\}\ [\![v]\!]\ \overline{(\mathsf{c}\ \{\mathsf{a}\}\ \mathsf{b}\ \mathsf{u})}\ [\![\mathsf{c}\ \{\mathsf{a}\}\ \mathsf{b}\ \mathsf{u}]\!]\end{aligned}$$

Because our translation is syntactic, we must discuss whether the constructed inductive family is well-founded. There is more than one syntactic criterion that ensures that a family is well-founded. It is beyond the scope of this paper to discuss the merits of each criterion. We pick the following one, which is, for example, used in the Agda system. If recursive occurrences of the type occur only in strictly positive positions in the type of the arguments of its constructors, then the family is well-founded. Because our translation preserves polarities, it preserves well-foundedness, according to the above criterion.

From this we deduce that the deductive translation is well-founded as well. Indeed, the eliminator has the same type in both the cases (considering the type of the inductive family itself as abstract), and its reduction rules are also identical.

6 Internalisation

We know that free theorems hold for any term of the PTS S (and these theorems are expressible and provable in S^r). Unfortunately, users of the logical system S^r which reflects S cannot take advantage of that fact: they have to redo the proofs for every new program (even though the proof is derivable, thanks to $[\![\_]\!]$). We would like the instances of the abstraction theorem to come truly for free: that is, extend S^r with a suitable construct that makes parametricity arguments available for every program

in S. To do so, we construct a new system S_p^r, which is the system S^r extended with following axiom schema.

Axiom 6.1 (*parametricity*)
For every closed type B of sort s ($\vdash_S B : s$), assume

$$\mathsf{param}_B : \forall^{k_i} \mathsf{x} : B.\ [\![B]\!]\ \mathsf{x} \ldots \mathsf{x}$$

The consistency of the new system remains to be shown. This can be done via a sound translation from S_p^r to S^r. The first attempt would be to extend to do so by translating $\mathsf{param}_B\ A$ into $[\![A]\!]$. Unfortunately, the above fails if A is an open term, because $[\![A]\!]$ contains occurrences of the variable x_R, which is not bound in the context of $\mathsf{param}_B\ A$. Therefore, we need a more complex interpretation. Even with a more complex interpretation accounting for free variables in A, we need to stick to closed types. Indeed, if the type B were to contain free variables, the type of param_B would not be well-scoped.

Parametricity witnesses Our attempt to show consistency by giving a local interpretation of the parametricity principle failed. Therefore, we instead can do a "global" transformation of a closed term in S_p^r to a term in S^r.

The idea is to transform the program such that, whenever a variable $(\mathsf{x} : A)$ is bound, a witness $(\mathsf{x}_\mathsf{R} : [\![A]\!]\ \mathsf{x} \ldots \mathsf{x})$ that x satisfies the parametricity condition is bound at the same time. Thus, functions are modified to take an additional argument witnessing that the original arguments are parametric. This additional argument is used to interpret occurrences of x in the argument of param_B. At every application, the parametricity witness can be reconstructed using the $[\![\_]\!]$ translation of the original argument. For example, the context

$$\begin{aligned}
&\mathsf{Nat} : \star,\\
&\mathsf{suc} : \mathsf{Nat} \to \mathsf{Nat},\\
&\mathsf{m} : \mathsf{Nat},\\
&\mathsf{X} : \widetilde{\star},\\
&\mathsf{p} : \mathsf{Nat} \to \mathsf{X}
\end{aligned}$$

would be translated to:

$$\begin{aligned}
&\mathsf{Nat} : \star, && [\![\mathsf{Nat}]\!] : \mathsf{Nat} \to \mathsf{Nat} \to \widetilde{\star},\\
&\mathsf{suc} : \mathsf{Nat} \to \mathsf{Nat}, && [\![\mathsf{suc}]\!] : [\![\ \mathsf{Nat} \to \mathsf{Nat}\ ]\!]\ \mathsf{suc}\ \mathsf{suc},\\
&\mathsf{m} : \mathsf{Nat}, && [\![\mathsf{m}]\!] : [\![\ \mathsf{Nat}\ ]\!]\ \mathsf{m}\ \mathsf{m},\\
&\mathsf{X} : \widetilde{\star},\\
&\mathsf{p} : (\mathsf{n} : \mathsf{Nat}) \to [\![\mathsf{Nat}]\!]\ \mathsf{n}\ \mathsf{n} \to \mathsf{X}
\end{aligned}$$

The term $\mathsf{p}\ (\mathsf{suc}\ \mathsf{m})$ is typeable in the source context, and would be translated to the term $\mathsf{p}\ (\mathsf{suc}\ \mathsf{m})\ ([\![\mathsf{suc}\ \mathsf{m}]\!])$. In the same context, $\mathsf{param}_{\mathsf{Nat}}\ \mathsf{m}$ would merely be translated to $[\![\mathsf{m}]\!]$.

General case In the rest of the section, we define the translation $\langle\!|\_|\!\rangle$ from terms of S_p^r to terms of S^r. The translation is similar to $[\![\_]\!]$, with a number of differences:

- The new translation deals with a richer language: There is a structure in the space of sorts, which can be either of the form s or $\widetilde{s}$. Further, it does not duplicate the bindings whose types are not in the source language (the sort is of the form $\widetilde{s}$). Therefore, it behaves differently depending on this sort, and using sorts, we must therefore distinguish two parts of the PTS: one (the source language of $[\![\_]\!]$), which deals with programs and types of sort s, and another that deals with parametricity proofs and propositions of sort $\widetilde{s}$ (the target language).
- The translation does not transform types to relations.
- The new translation does not replicate the bindings: It adds at most one additional binding, regardless of the arity of param. A consequence is that the renaming operation (Definition 3.1) must be modified such that occurrences of variables bound in bindings processed by $\langle\!|\_|\!\rangle$ are not renamed.

As hinted above, $\langle\!|\_|\!\rangle$ does not work on all possible system S^r. The precise set of restrictions is as follows.

Definition 6.2 (*Restrictions for internalisation*)

1. Let $\widetilde{\mathbb{S}} = \mathbb{S}^r - \mathbb{S}$. If $s \in \mathbb{S}$, then $\widetilde{s} \in \widetilde{\mathbb{S}}$. This ensures that the sorts of types of the sources language can always be distinguished from the sorts of propositions.[3]
2. If $(k, s_1, s_2, s_3) \in \mathbb{R}^r$ and $s_3 \in \mathbb{S}$, then $s_1 \in \mathbb{S}$ and $s_2 \in \mathbb{S}$. This ensures that terms and types of the source language can contain no propositions of parametricity nor their proofs.
3. Let $K_v \subseteq K$ and $K_w = K - K_v$. (In the following, we will use the meta-syntactic variable a for colours in the first group and b for colours in the second one.) If $(k, s_1, s_2, s_3) \in \mathbb{R}$ then $s_1 \in \mathbb{S} \leftrightarrow k \in K_v$.
   This ensures that quantifications over terms in the input language can be recognised syntactically from quantifications over parametricity propositions and proofs. This requirement is for convenience only, as suitable colours can be inferred from a typing derivation.
4. For each rule $s_1 \overset{v}{\rightsquigarrow} s_2$ there must be a colour $t_v \in K_w$ and a rule $\widetilde{s_1} \overset{t_v}{\rightsquigarrow} \widetilde{s_2}$.

For example, the system described in Section 3.3 satisfies these conditions.

In the following, we assume that param_B is always saturated. Doing so causes no loss of generality: η-expansion can be applied to obtain the desired form. We define the translation $\langle\!|\_|\!\rangle$ from terms typed in S_p^r to terms of S^r as follows.

[3] This restriction rules out (non-trivial) reflective systems.

Definition 6.3 (*Compilation of* param)

$$\begin{aligned}
\langle\!|s|\!\rangle &= s \\
\langle\!|\mathsf{x}|\!\rangle &= \mathsf{x} \\
\langle\!|\mathsf{param}_B\ F\ A_0 \ldots A_l|\!\rangle &= [\![F]\!]\ A_0 \ldots A_l \\
\hline
\langle\!|(\mathsf{x}:A) \xrightarrow{v} B|\!\rangle &= (\mathsf{x}:A) \xrightarrow{v} (\mathsf{x}_\mathsf{R}:[\![A]\!]\ \mathsf{x} \ldots \mathsf{x}) \xrightarrow{t_v} \langle\!|B|\!\rangle \\
\langle\!|\lambda^v \mathsf{x}:A.\ b|\!\rangle &= \lambda^v \mathsf{x}:A.\ \lambda^{t_v} \mathsf{x}_\mathsf{R}:[\![A]\!]\ \mathsf{x} \ldots \mathsf{x}.\ \langle\!|b|\!\rangle \\
\langle\!|F \bullet_v a|\!\rangle &= \langle\!|F|\!\rangle \bullet_v a \bullet_{t_v} [\![a]\!] \qquad (\dagger) \\
\hline
\langle\!|(\mathsf{x}:A) \xrightarrow{w} B|\!\rangle &= (\mathsf{x}:\langle\!|A|\!\rangle) \xrightarrow{w} \langle\!|B|\!\rangle \\
\langle\!|\lambda^w \mathsf{x}:A.\ b|\!\rangle &= \lambda^w \mathsf{x}:\langle\!|A|\!\rangle.\ \langle\!|b|\!\rangle \\
\langle\!|F \bullet_w a|\!\rangle &= \langle\!|F|\!\rangle \bullet_w \langle\!|a|\!\rangle \qquad (*) \\
\hline
\langle\!|\Gamma, \mathsf{x}:A|\!\rangle &= \langle\!|\Gamma|\!\rangle, \mathsf{x}:A, \mathsf{x}_\mathsf{R}:[\![A]\!]\ \mathsf{x} \ldots \mathsf{x} && \text{if } \Gamma \vdash A : s \\
\langle\!|\Gamma, \mathsf{x}:A|\!\rangle &= \langle\!|\Gamma|\!\rangle, \mathsf{x}:\langle\!|A|\!\rangle && \text{if } \Gamma \vdash A : \widetilde{s}
\end{aligned}$$

Lemma 6.4
Assuming $s \in \mathbb{S}$, then

1. if $\Gamma \vdash_{S^r} B : s$, then param cannot appear in B and
2. if $\Gamma \vdash_{S^r} A : B$, then param cannot appear in A.

Proof
The proof is done by simultaneous induction on typing derivations.

- In the base case, a constant cannot be param, because its type has a sort of form $\widetilde{s}$, which is distinct from s, by assumption 1 in Definition 6.2.
- In the induction cases, we take advantage of restriction 2 in Definition 6.2 to ensure that subterms also satisfy the conditions of the lemma. □

Theorem 6.5
All occurrences of param are removed by $\langle\!|\_|\!\rangle$.

Proof
The proof is done by induction on terms.

- The base case (param_B) removes occurrences.
- No other occurrences are introduced. In particular, in the line marked with an asterisk (*); the argument of sort $\widetilde{s}$ (which may contain param) is not duplicated. In line marked (†), the term a cannot contain any occurrence of param, as shown by Lemma 6.4. □

Theorem 6.6 (*soundness*)
$\langle\!|\_|\!\rangle$ translates valid judgements in S^r_p to valid judgements in S^r,

$$\Gamma \vdash_{S^r_p} A : B \Rightarrow \langle\!|\Gamma|\!\rangle \vdash_{S^r} \langle\!|A|\!\rangle : \langle\!|B|\!\rangle$$

Proof sketch
The proof proceeds by induction on the typing derivation. □

7 Applications

Sections 3 and 5 contain simple applications of our setting. In this section we see how elaborate constructions can be handled. All examples of this section fit within the system I_ω augmented with inductive definitions.

7.1 A library for applications

Applying $[\![\_]\!]$ by hand to non-trivial examples can be tedious. The reader eager to experiment is suggested to use computer aids. One possible tool is that of Böhme (2007), which computes the relational interpretation of any Haskell type. Unfortunately, the above tool has not been extended to support dependent types. To generate the examples for this paper, we have used an Agda library (Bernardy 2010) instead. An advantage of the library approach is that one can use a single framework to write programs and reason using free theorems about them.

7.2 Proof irrelevance and parametricity

In this section we show that any two proofs of a given proposition can be treated as related. In a predicative system with inductive families, such as Agda, there are at least two ways to represent propositions. A common choice is to use $\star$ for the sort of propositions, as we have suggested in Section 2.1. One issue is then that quantification over types in $\star$ is in $\star_1$, hence not a proposition. The issue can be side-stepped by encoding propositions in a universe like the following Prop, where quantification using π yields a proposition in the Prop universe,

```
data Prop : ⋆₁ where
  top : Prop
  bot : Prop
  _∧_ : Prop → Prop → Prop
  π   : (A : ⋆) → (f : A → Prop) → Prop
```

One can then construct proposition objects, for example a usual ordering between naturals

```
_⩽_ : Nat → Nat → Prop
zer   ⩽ n     = top
suc m ⩽ zer   = bot
suc m ⩽ suc n = m ⩽ n
```

or the predicate that n is the biggest natural:

```
supremum : Nat → Prop
supremum n = π Nat (λ m → m ⩽ n)
```

The intention is for propositions to be interpreted as the set of their proofs. The following function realises this interpretation in the standard way: truth is interpreted as a singleton type, falsity as an empty type, intersection of propositions as a pair of proofs, and quantification as a product.

$$
\begin{aligned}
&\mathsf{Proof} : \mathsf{Prop} \to \star \\
&\mathsf{Proof\ top} \quad = \top \\
&\mathsf{Proof\ bot} \quad = \bot \\
&\mathsf{Proof}\ (\mathsf{a} \wedge \mathsf{b}) \ = \mathsf{Proof\ a} \times \mathsf{Proof\ b} \\
&\mathsf{Proof}\ (\pi\ \mathsf{A\ f}) \ = (\mathsf{a} : \mathsf{A}) \to \mathsf{Proof}\ (\mathsf{f\ a})
\end{aligned}
$$

However, to enable changing the parametricity translation of proofs, we will instead just postulate an abstract $\mathsf{Proof} : \mathsf{Prop} \to \star$ and a few constants, chosen so that proofs (terms of type $\mathsf{Proof\ p}$ for some $\mathsf{p} : \mathsf{Prop}$) only can interact in limited ways with programs ($\mathsf{a} : \mathsf{A} : \star$). We allow standard proof constructions: introduction (abs) and elimination (app) of π, introduction (pair) and elimination (proj_1,proj_2) of, $\wedge$ and introduction (obvious) of top. In addition, given any proof of falsity, a program of an arbitrary type can be constructed (using botElim). By seeing the arguments as premisses and the results as conclusions, one recognises the standard inference rules in the types of these constants

$$
\begin{aligned}
&\mathsf{app} : (\mathsf{A} : \star) \to (\mathsf{f} : \mathsf{A} \to \mathsf{Prop}) \to \mathsf{Proof}\ (\pi\ \mathsf{A\ f}) \to (\mathsf{a} : \mathsf{A}) \to \mathsf{Proof}\ (\mathsf{f\ a}) \\
&\mathsf{abs} : (\mathsf{A} : \star) \to (\mathsf{f} : \mathsf{A} \to \mathsf{Prop}) \to ((\mathsf{a} : \mathsf{A}) \to \mathsf{Proof}\ (\mathsf{f\ a})) \to \mathsf{Proof}\ (\pi\ \mathsf{A\ f}) \\
&\mathsf{proj}_1 : (\mathsf{a\ b} : \mathsf{Prop}) \to \mathsf{Proof}\ (\mathsf{a} \wedge \mathsf{b}) \to \mathsf{Proof\ a} \\
&\mathsf{proj}_2 : (\mathsf{a\ b} : \mathsf{Prop}) \to \mathsf{Proof}\ (\mathsf{a} \wedge \mathsf{b}) \to \mathsf{Proof\ b} \\
&\mathsf{pair} : (\mathsf{a\ b} : \mathsf{Prop}) \to \mathsf{Proof\ a} \to \mathsf{Proof\ b} \to \mathsf{Proof}\ (\mathsf{a} \wedge \mathsf{b}) \\
&\mathsf{obvious} : \mathsf{Proof\ top} \\
&\mathsf{botElim} : \mathsf{Proof\ bot} \to (\mathsf{A} : \star) \to \mathsf{A}
\end{aligned}
$$

A consequence of restricting oneself to an abstract representation of proofs is that the structure of proofs is *irrelevant* in the meaning of programs. The reason is that programs cannot assume that the structure of a proof corresponds that of the proposition being examined in any way.

Note that programs could depend on the structure of proofs if we were to use the *definition* of Proof given above, and in that case our relational interpretation would translate proofs to witnesses that these are related. For example, given the type of a lookup function in a list bound by length

$$\mathsf{lk} : \{\mathsf{A} : \star\} \to (\mathsf{n} : \mathsf{Nat}) \to (\mathsf{xs} : \mathsf{List\ A}) \to \mathsf{Proof}\ (\mathsf{n} < \mathsf{len\ xs}) \to \mathsf{A}$$

one gets the following relation, which carries an assumption $\mathsf{p_R}$ requiring the proofs p_1 and p_2 to be related. That assumption would have a complicated formulation if we had taken the standard interpretation of the set of proofs,

$$
\begin{aligned}
[\![\mathsf{lk}]\!] \ : \ & \{\mathsf{A}_1\ \mathsf{A}_2 : \star\}\ (\mathsf{A_R} : \mathsf{A}_1 \to \mathsf{A}_2 \to \star) \\
& \{\mathsf{n}_1\ \mathsf{n}_2 : \mathsf{Nat}\}\ (\mathsf{n_R} : [\![\mathsf{Nat}]\!]\ \mathsf{n}_1\ \mathsf{n}_2) \\
& \{\mathsf{xs}_1 : \mathsf{List\ A}_1\}\ \{\mathsf{xs}_2 : \mathsf{List\ A}_2\}\ (\mathsf{xs_R} : [\![\mathsf{List}]\!]\ \mathsf{A_R}\ \mathsf{xs}_1\ \mathsf{xs}_2) \\
& \{\mathsf{p}_1 : \mathsf{Proof}\ (\mathsf{n}_1 < \mathsf{len\ xs}_1)\} \\
& \{\mathsf{p}_2 : \mathsf{Proof}\ (\mathsf{n}_2 < \mathsf{len\ xs}_2)\} \\
& (\mathsf{p_R} : [\![\mathsf{Proof}]\!]\ [\![\mathsf{n} < \mathsf{len\ xs}]\!]\ \mathsf{p}_1\ \mathsf{p}_2) \to \\
& \mathsf{A_R}\ (\mathsf{lk}\ \mathsf{n}_1\ \mathsf{xs}_1\ \mathsf{p}_1)\ (\mathsf{lk}\ \mathsf{n}_1\ \mathsf{xs}_1\ \mathsf{p}_1)
\end{aligned}
$$

However, by axiomatising Proof, we can pick any translation ⟦Proof⟧ that also satisfies other axioms. In fact, we can assert that all proofs are related:

$$\begin{aligned}&⟦\mathsf{Proof}⟧ : ⟦\mathsf{proposition} \to *⟧\ \mathsf{Proof}\ \mathsf{Proof}\\ &⟦\mathsf{Proof}⟧\ \_\ x_1\ x_2 = \top\end{aligned}$$

The assumptions requiring proofs to be related then reduce to ⊤; effectively disappearing (because values of singleton types like ⊤ can always be inferred).

For the above overriding to be sound, one needs to provide a translation of app, abs, proj_1, proj_2, pair, obvious, and botElim respecting the parametricity condition. All but the last are easy to translate: their results are Proofs, so the result type of their translation is ⊤. Hence, constant functions returning tt do the job. Translating botElim can seem more tricky, because it has a proof as argument, the assertion that all proofs are related makes ⟦botElim⟧ potentially more difficult to write, as it has one less assumption to work with. However, because botElim already has a proof of falsity as an argument, its translation has two of them. Hence, one can prove anything ⟦botElim⟧ by using them, making the relational witness superfluous,

$$\begin{aligned}&⟦\mathsf{botElim}⟧ : (b_1 : \mathsf{Proof\ bot}) \to (b_2 : \mathsf{Proof\ bot}) \to \top \to\\ &\qquad ⟦(A : \star) \to A⟧\ (\mathsf{botElim}\ b_1)\ (\mathsf{botElim}\ b_2)\\ &⟦\mathsf{botElim}⟧\ b_1\ b_2 = \mathsf{botElim}\ b_1\ (⟦(A : \star) \to A⟧\ (\mathsf{botElim}\ b_1)\ (\mathsf{botElim}\ b_2))\end{aligned}$$

In summary, assuming proof-irrelevance, proof arguments do not strengthen parametricity conditions in useful ways. One often (but not always) does not care about the *actual* proof of a proposition, but merely that it exists. In that case, knowing that two proofs are related adds no information.

7.3 Type classes

What if a function is not parametrised over all types, but only types equipped with decidable equality? One way to model this difference in a PTS is to add an extra parameter to capture the extra constraint. For example, a function nub : Nub removing duplicates from a list may be given the following type:

$$\mathsf{Nub} = (A : \star) \to \mathsf{Eq}\ A \to \mathsf{List}\ A \to \mathsf{List}\ A$$

The equality requirement itself may be modelled as a mere comparison function: $\mathsf{Eq}\ A = A \to A \to \mathsf{Bool}$. In that case, the parametricity statement is amended with an extra requirement on the relation between types, which expresses that eq_1 and eq_2 must respect the A_R relation. Formally:

$$\begin{aligned}&⟦\mathsf{Eq}\ A⟧\ \mathsf{eq}_1\ \mathsf{eq}_2 = \{a_1 : A_1\} \to \{a_2 : A_2\} \to A_R\ a_1\ a_2 \to\\ &\qquad \{b_1 : A_1\} \to \{b_2 : A_2\} \to A_R\ b_1\ b_2 \to\\ &\qquad ⟦\mathsf{Bool}⟧\ (\mathsf{eq}_1\ a_1\ b_1)\ (\mathsf{eq}_2\ a_2\ b_2)\\ &⟦\mathsf{Nub}⟧\ n_1\ n_2 =\\ &\quad \{A_1\ A_2 : \star\} \to (A_R : ⟦\star⟧\ A_1\ A_2) \to\\ &\quad \{\mathsf{eq}_1 : \mathsf{Eq}\ A_1\} \to \{\mathsf{eq}_2 : \mathsf{Eq}\ A_2\} \to ⟦\mathsf{Eq}\ A⟧\ \mathsf{eq}_1\ \mathsf{eq}_2 \to\\ &\quad \{\mathsf{xs}_1 : \mathsf{List}\ A_1\} \to \{\mathsf{xs}_2 : \mathsf{List}\ A_2\} \to ⟦\mathsf{List}\ A⟧\ \mathsf{xs}_1\ \mathsf{xs}_2 \to\\ &\quad ⟦\mathsf{List}⟧\ A_R\ (n_1\ A_1\ \mathsf{eq}_1\ \mathsf{xs}_1)\ (n_2\ A_2\ \mathsf{eq}_2\ \mathsf{xs}_2)\end{aligned}$$

So far, this is just confirming the informal description in Wadler (1989). But with access to full dependent types, one might wonder, what if we model equality more precisely, for example, by requiring eq to be reflexive?

$$\begin{aligned}\mathsf{Eq'\ A} &= (\mathsf{eq} : \mathsf{A} \to \mathsf{A} \to \mathsf{Bool}) \times \mathsf{Refl\ eq}\\ \mathsf{Refl\ eq} &= (\mathsf{x} : \mathsf{A}) \to \mathsf{eq\ x\ x} \equiv \mathsf{true}\end{aligned}$$

In the case of Eq′, the parametricity condition does not become more exciting. It merely requires the proofs of reflexivity at A_1, A_2 to be related. This extra condition adds nothing new, as seen in Section 7.2.

The observations drawn from this simple example can be generalised: type-classes may be encoded as their dictionary of methods (Wadler & Blott 1989), ignoring their laws. Indeed, even if a type class has associated laws, they have little impact on the parametricity results.

7.4 Constructor classes

Having seen how to apply our framework both to type constructors and type classes, we now apply it to types quantified over a type constructor, with constraints.

Voigtländer (2009) provides many such examples, using the Monad constructor class. They fit well in our framework, but here we show the simpler example of Functors, which already captures the essence of constructor classes,

$$\begin{aligned}&\mathsf{Functor} : \star_1\\ &\mathsf{Functor} = (\mathsf{F} : \star \to \star) \times ((\mathsf{X\ Y} : \star) \to (\mathsf{X} \to \mathsf{Y}) \to \mathsf{F\ X} \to \mathsf{F\ Y})\end{aligned}$$

Our translation readily applies to the above definition, and yields the following relation between functors:

$$\begin{aligned}&[\![\mathsf{Functor}]\!] : \mathsf{Functor} \to \mathsf{Functor} \to \star_1\\ &[\![\mathsf{Functor}]\!]\ (\mathsf{F}_1, \mathsf{map}_1)\ (\mathsf{F}_2, \mathsf{map}_2)\\ &\quad = (\mathsf{F_R} : \{\mathsf{A}_1\ \mathsf{A}_2 : \star\} \to (\mathsf{A_R} : \mathsf{A}_1 \to \mathsf{A}_2 \to \star) \to (\mathsf{F}_1\ \mathsf{A}_1 \to \mathsf{F}_2\ \mathsf{A}_2 \to \star)) \times\\ &\qquad (\ \{\mathsf{X}_1\ \mathsf{X}_2 : \star\} \to (\mathsf{X_R} : \mathsf{X}_1 \to \mathsf{X}_2 \to \star) \to\\ &\qquad\ \{\mathsf{Y}_1\ \mathsf{Y}_2 : \star\} \to (\mathsf{Y_R} : \mathsf{Y}_1 \to \mathsf{Y}_2 \to \star) \to\\ &\qquad\ \{\mathsf{f}_1 : \mathsf{X}_1 \to \mathsf{Y}_1\} \to \{\mathsf{f}_2 : \mathsf{X}_2 \to \mathsf{Y}_2\} \to\\ &\qquad\ (\{\mathsf{x}_1 : \mathsf{X}_1\} \to \{\mathsf{x}_2 : \mathsf{X}_2\} \to \mathsf{X_R\ x}_1\ \mathsf{x}_2 \to \mathsf{Y_R\ (f}_1\ \mathsf{x}_1)\ (\mathsf{f}_2\ \mathsf{x}_2)) \to\\ &\qquad\ \{\mathsf{y}_1 : \mathsf{F}_1\ \mathsf{X}_1\} \to \{\mathsf{y}_2 : \mathsf{F}_2\ \mathsf{X}_2\} \to (\mathsf{y_R} : \mathsf{F_R\ X_R\ y}_1\ \mathsf{y}_2) \to\\ &\qquad\ \mathsf{F_R\ Y_R\ (map}_1\ \mathsf{f}_1\ \mathsf{y}_1)\ (\mathsf{map}_2\ \mathsf{f}_2\ \mathsf{y}_2)\ )\end{aligned}$$

In words, the translation of a functor is the product of a relation transformer (F_R) between functors F_1 and F_2, and a witness that map_1 and map_2 preserve relations.

Such Functors can be used to define a generic fold operation, which typically takes the following form:

$$\begin{aligned}&\textbf{data}\ \mu\ ((\mathsf{F}, \mathsf{map}) : \mathsf{Functor}) : \star\ \textbf{where}\\ &\quad \mathsf{In} : \mathsf{F}\ (\mu\ (\mathsf{F}, \mathsf{map})) \to \mu\ (\mathsf{F}, \mathsf{map})\\ &\mathsf{fold} : ((\mathsf{F}, \mathsf{map}) : \mathsf{Functor}) \to (\mathsf{A} : \star) \to (\mathsf{F\ A} \to \mathsf{A}) \to \mu\ (\mathsf{F}, \mathsf{map}) \to \mathsf{A}\\ &\mathsf{fold}\ (\mathsf{F}, \mathsf{map})\ \mathsf{A}\ \phi\ (\mathsf{In\ d}) = \phi\ (\mathsf{map}\ (\mu\ (\mathsf{F}, \mathsf{map}))\ \mathsf{A}\ (\mathsf{fold}\ (\mathsf{F}, \mathsf{map})\ \mathsf{A}\ \phi)\ \mathsf{d})\end{aligned}$$

Note that the μ datatype is not strictly positive, so its use would be prohibited in many dependently typed languages to avoid inconsistency. However, if one restricts oneself to well-behaved functors (yielding strictly positive types), then consistency is restored both in the source and target systems, and the parametricity condition derived for fold is valid. One way to implement this restriction is to use containers, as defined by Morris & Altenkirch (2009).

One can see from the type of fold that it behaves uniformly over (F, map) as well as over A. By applying $[\![\_]\!]$ to fold and its type, this observation can be expressed (and justified) formally and used to reason about fold. Further, every function defined using fold, and in general any function parametrised over any functor, enjoys the same kind of property.

Gibbons & Paterson (2009) previously made a similar observation in a categorical setting, showing that fold is a natural transformation between higher order functors. Their argument heavily relies on categorical semantics and the universal property of fold, while our type-theoretical argument uses the type of fold as a starting point and directly obtains a parametricity property. However, some additional work is required to obtain the equivalent property using natural transformations and horizontal compositions from the parametricity property.

7.5 Type equality

Equality between types A and B can be expressed by the following relation, named after Leibniz, which asserts that any proof involving A can be converted to a proof involving B.

```
Equal : ⋆ → ⋆ → ⋆₁
Equal A B = (P : ⋆ → ⋆) → P A → P B
```

An intuitive reading of the type of Equal suggests that inhabitants of that type can only be polymorphic identity functions. Indeed, conversions from P A to P B, for an arbitrary P, cannot depend on actual values. We would like to apply the axiom of parametricity to recover a formal proof of that result.

Before doing so, we will do a practice round on the similar, but simpler, problem of showing that functions of type Id must be (extensionally) the identity function,

```
Id = (A : ⋆) → A → A
```

Using parametricity with arity $n = 1$, and taking advantage of the axiom schema introduced in Section 6, we have

```
paramId : (f : Id) →
          {A : Set} (AR : A → Set)
          {x : A} → (xR : AR x) →
          AR (f A x)
```

Then we can instantiate A_R with the predicate of "being equal to x, the input of f"; and its proof x_R with reflexivity of equality to obtain the desired result,

```
theorem : (f : Id) → (A : ⋆) → (x : A) → x ≡ f A x
theorem f A x = param_Id f (_≡_ x) refl
```

The proof of our original proposition follows the same pattern, with a single complication. Because Equal A B is an open term, our parametricity axiom cannot be applied to it directly. There is a simple trick that allows us to proceed though: bind the variables in a dependent pair and apply the axiom to that type. Parametricity then gives us

```
SomeEqual = (A : ⋆) × (B : ⋆) × Equal A B
param_SomeEqual : (s : SomeEqual) → ⟦SomeEqual⟧ s
```

where

```
⟦Equal⟧ {A} A_R {B} B_R = λ (e : Equal A B) →
  {P : ⋆ → ⋆} → (P_R : {X : Set} → (X → Set) → P X → Set)
  {p : P A} → P_R A_R x_1 →
  P_R B_R (e f p)
⟦SomeEqual⟧ (A, B, e) =
  (A_R : A → ⋆_1) ×
  (B_R : B → ⋆_1) ×
  (⟦Equal⟧ A_R B_R e)
```

Using this instantiation of the parametricity axiom, we can proceed as in the Id case, with three differences:

- The instantiation of the predicate constructor P_R takes an extra argument p, which we ignore.
- Because the input and output type are syntactically different, we use heterogeneous equality (_≅_), which is similar to _≡_, but relates values of different types.
- We ignore the predicates A_R and B_R constructed by param_ in the record of type ⟦SomeEqual⟧.

```
theorem : ∀ (A B : ⋆) → (e : Equal A B) → (P : ⋆ → ⋆) (x : P A) → x ≅ e Px
theorem A B e P x = q
  where (_, _, q) = (param_SomeEqual (A, B, e) {P} (λ p → ((_≅_) x)) refl)
```

Some points are worth emphasising:

- It is possible to get a result about an open term, even though our axiom only handles closed terms. Still, we get a concrete result (the above theorem) that does not involve any occurrence of the parametricity axiom. This happens because the function constructing predicates (λ p → ((_≅_) x)) precisely discards those occurrences.
- The result is already exposed by Vytiniotis & Weirich (2010), but it is remarkable that its proof is one line long given our framework.
- Because the equality _≅_ is heterogeneous, deriving a substitution principle from it requires Streicher's Axiom K (Hofmann & Streicher 1996).

In consequence, it seems that one cannot derive that all proofs of equality are equal from the axiom of parametricity.

8 Discussion

8.1 Related work

Studies of parametricity for System F and its variants abound in the literature, starting with the seminal paper by Reynolds (1983), where the polymorphic semantics of System Ftypes is captured in a suitable model.

We use here a more syntactic approach, where the expressions of the programming language are (syntactically) translated to formulas describing the program. This style was pioneered by Mairson (1991) and used by a number of authors, including Abadi et al. (1993), Plotkin & Abadi (1993), and Wadler (2007). In particular, Wadler (2007) gives an insightful presentation of the abstraction theorem, as the inverse of Girard's (1972) Representation theorem: Reynolds (1983) gives an embedding from System F to second-order logic, while Girard (1972) gives the corresponding projection. Our version of the abstraction theorem differs in the following aspects from that of Wadler (2007) (and to our knowledge all others):

1. Instead of targeting a logic, we target its *propositions-as-types* interpretation, expressed in a PTS.
2. We abstract from the details of the systems, generalising to a class of PTSs.
3. We add that the translation function used to interpret types as relations can also be used to interpret terms as witnesses of those relations. In short, the $[\![A]\!]$ part of $\Gamma \vdash A : B \implies [\![\Gamma]\!] \vdash [\![A]\!] : [\![B]\!]\ \overline{A}$ is new. This additional insight depends heavily on using the interpretation of propositions as types.

The question of how Girard's projection generalises to arbitrary PTSs naturally arises, and is addressed by Bernardy & Lasson (2011).

One direction of research is concerned with parametricity in extensions of System F. Our work is directly inspired by Vytiniotis & Weirich (2010), which extend parametricity to (an extension of) Fω: indeed, Fω can be seen as a PTS with one more product rule than System F.

Before that, Takeuti (2004, personal communication) attempted to extend CC with parametricity. He asserted parametricity at all types in a similar way as we do here, in fact extending similar axiom schemes for System F by Plotkin & Abadi (1993). For each $\alpha : \Box$ and $P : \alpha$, Takeuti (2004, personal communication) defined a relational interpretation $\langle P \rangle$ and a kind $(\!|P : \alpha|\!)$ such that $\langle P \rangle : (\!|P : \alpha|\!)$. Then for each type $T : \star$, he postulated an axiom $\mathsf{param}_T : (\forall x : T.\ \langle T \rangle\ x\ x)$, conjecturing that such axioms did not make the system inconsistent. For closed terms P, Takeuti's translations $\langle P \rangle$ and $(\!|P : \alpha|\!)$ resemble our $[\![P]\!]$ and $[\![\alpha]\!]\ \overline{P}$, respectively, (with $n = 2$), but the pattern is obscured by an error in the translation rule for the product $\Box \rightsquigarrow \star$. His omission of a witness $\mathsf{x_R}$ for the relationship between values x_1 and x_2 in the rules corresponding to the product $\star \rightsquigarrow \Box$ appears to correspond to a computationally irrelevant interpretation of $\star$, as we present in Section 7.2.

In previous work (Bernardy *et al.* 2010) we have shown that the relational interpretation can be generalised to PTSs. Here we extend the results in multiple ways:

- We have annotated the relational interpretation with colours, clarifying the role of each type of quantification, and showing how the translation can take advantage of systems with implicit syntax (Section 4).
- We have proven that our previous inductive relational interpretation of inductive families is correct (Section 5.4).
- We have shown that part of the meta-theory of parametricity can be internalised into a PTS and that the theory remains consistent (for an important class of systems) (Section 6).
- We have argued in detail, why one can assume that two proofs of a given proposition are always related (Section 7.2).
- We have shown in an example that the support of Σ types allows us to get results for open types, even with an axiom schema restricted to closed types (Section 7.5).
- We allow for the source and target system to be different.

Bernardy & Lasson (2011) have shown how to construct a logic for parametricity for an arbitrary source PTS (Definition 3.3), which is as consistent as the source PTS.

Besides supporting more sorts and function spaces, an orthogonal extension of the parametricity theory is to support impure features in the system. For example, Johann & Voigtländer (2006) studied how explicit strictness modifies parametricity results. It is not obvious how to support such extensions in our framework.

It also appears that the function $[\![\_]\!]$ (for the unary case) has been discovered independently by Monnier & Haguenauer (2010) for a very different purpose. They use $[\![\_]\!]$ as a compilation function from CC to a language with singleton types as the sole way to express dependencies from values to types. Their goal is to enforce phase-distinction between compile-time and run-time. Type preservation of the translation scheme is the main formal property presented by Monnier & Haguenauer (2010). We remark that this property is a specialisation of our abstraction theorem for CC. Another lesson learnt from this parallel is that the unary $[\![\_]\!]$ generates singleton types.

8.2 Future work

Our explanation of parametricity for dependent types has opened a whole range of interesting topics for future work.

We should investigate whether our framework can be applied (and extended if need be) to more exotic systems, for example those incorporating strictness annotations (`seq`) or non-termination.

We gave an interpretation of the axiom of parametricity as a compilation pass to a language not requiring the axiom. It would also be interesting to, instead, extend the β-reduction rules to support the axiom.

The target PTS that we constructed has typed individuals, whereas many logics for parametricity have untyped individuals. Girard's (1972) representation theorem

shows that in System F such type of information can be recovered and is therefore not essential. It would be worthwhile to generalise that result to arbitrary PTSs.

We presented only simple examples. Applying the results to more substantial applications should be done as well. In particular, we hope that our results open the door to a more streamlined way of getting free theorems for domain-specific programming languages. One would proceed along the following steps:

1. model the domain-specific languages within a dependently typed language.
2. Use $[\![\_]\!]$ to obtain parametricity properties of any function of interest.
3. Prove domain-specific theorems, using parametricity properties.

We think that the above process is an economical way to work with parametricity for extended type systems. Indeed, developing languages with exotic-type systems as an embedding in a dependently typed language is increasingly popular (Oury & Swierstra 2008), and that is the first step in the above process. By providing an automatic second step, we hope to spare language designers the effort to adapt Reynolds' (1983) abstraction theorem for new type systems in an ad-hoc way. Indeed, Pouillard (2011) has derived correctness properties of a library for names and binders by following our method.

A Proof of the abstraction theorem

A.1 Proof outline

In this appendix we provide the proof of our main theorem.

Theorem A.1 (*abstraction*)
If the PTS S^r reflects S,

$$\Gamma \vdash_S A : B \implies [\![\Gamma]\!] \vdash_{S^r} [\![A]\!] : [\![B]\!]\ \overline{A}$$

Proof sketch
A derivation of $[\![\Gamma]\!] \vdash [\![A]\!] : [\![B]\!]\ \overline{A}$ in S^r is constructed by induction on the derivation of $\Gamma \vdash A : B$ in S, using the syntactic properties of PTSs. We have one case for each typing rule: each typing rule translates to a portion of a corresponding relational typing judgement, as shown in Figure A1. For each rule, the translation of the premises (induction hypotheses) and the conclusion (inductive conclusion) are presented on the right-hand column. The rest of the proof consists in building derivation trees linking the inductive hypotheses to the expected conclusion. At this point, filling the trees is mostly straightforward because the construction of the tree is guided by the syntax of the conclusion that we want to prove. Taking, for example, the case of product, the outline of the derivation tree is to use once the ABSTRACTION rule, then PRODUCT twice. For the abstraction case, the target derivation must use ABSTRACTION twice.

Once the outline is in place, filling in the details takes a lot of space, mainly for two reasons:

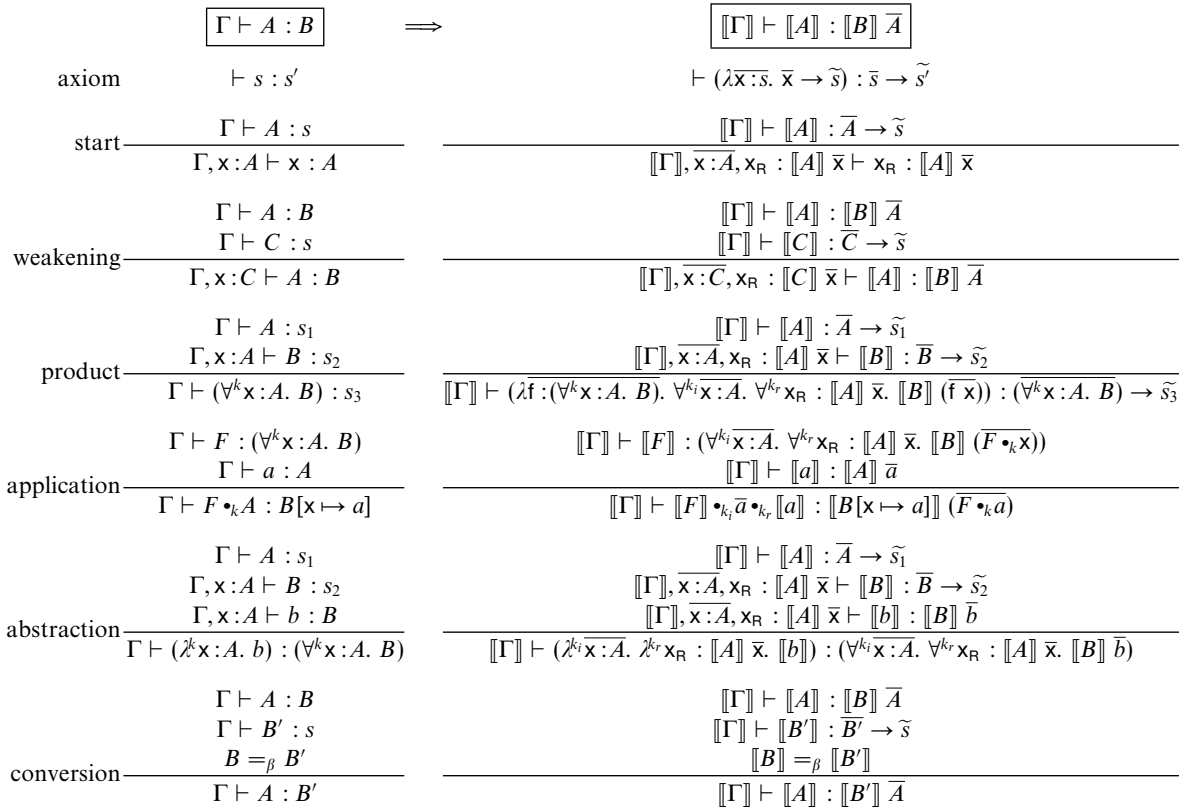


Fig. A.1. Outline of a proof of Theorem 3.12 by induction over the derivation of $\Gamma \vdash A : B$. In the left-hand column, rules of the typing judgement $\Gamma \vdash A : B$ are listed. For conciseness, a variant form of the abstraction rule is used in this outline; equivalence of the two systems follows from Barendregt (1992, Lemma 5.2.13). The conversion case uses Lemma 3.11.

1. Every time that translation generates a test that a value satisfies a relational interpretation, it generates a redex. (That is, the translation is not in normal form.) Typing such a redex is much more verbose than typing its normal form.
2. There is certain redundancy in the typing rules of PTS presented by Barendregt (1992). For example, to check an abstraction one must check that its type (a function) is well-sorted. It is, however, likely that the domain and co-domain of the product will have to be rechecked somewhere else in the tree. Some of these duplications have been factored below, but not all. □

Further proof details are provided on the following pages.

A.2 Proof details

The following propositions are proved by simultaneous induction on the typing judgement:

lem $\Gamma \vdash_S A : s \implies [\![\Gamma]\!] \vdash_{S^r} \overline{A} : \overline{s}$.
Proved by the thinning lemma (Barendregt 1992, Lemma 5.2.12, p. 220). For each A_i, erase from the context $[\![\Gamma]\!]$ the relational variables and j-indexed variables such that $j \neq i$. The legality of the context is ensured by **ind**.

ind $\Gamma \vdash_S A : B \implies [\![\Gamma]\!] \vdash_{S^r} [\![A]\!] : [\![B]\!]\ \overline{A}$.
The proof proceeds by induction on the derivation of $\Gamma \vdash A : B$. We have one case for each typing rule: each typing rule translates to a portion of a corresponding relational typing judgement; and we detail them in the rest of the section. The construction of the derivation makes use of the propositions **lem**, **ind**, and **ind'** (on smaller judgements).

ind' $\Gamma \vdash_S B : s \implies [\![\Gamma]\!] \vdash_{S^r} [\![B]\!] : \overline{B} \to \widetilde{s}$
Corollary of **ind**.

We proceed with the case analysis for the proof of **ind**.

axiom $\mathbf{c} : s$ If $\mathbf{c}$ is a sort, this follows from Lemma 3.8. Otherwise, the proposition is assumed as an hypothesis.

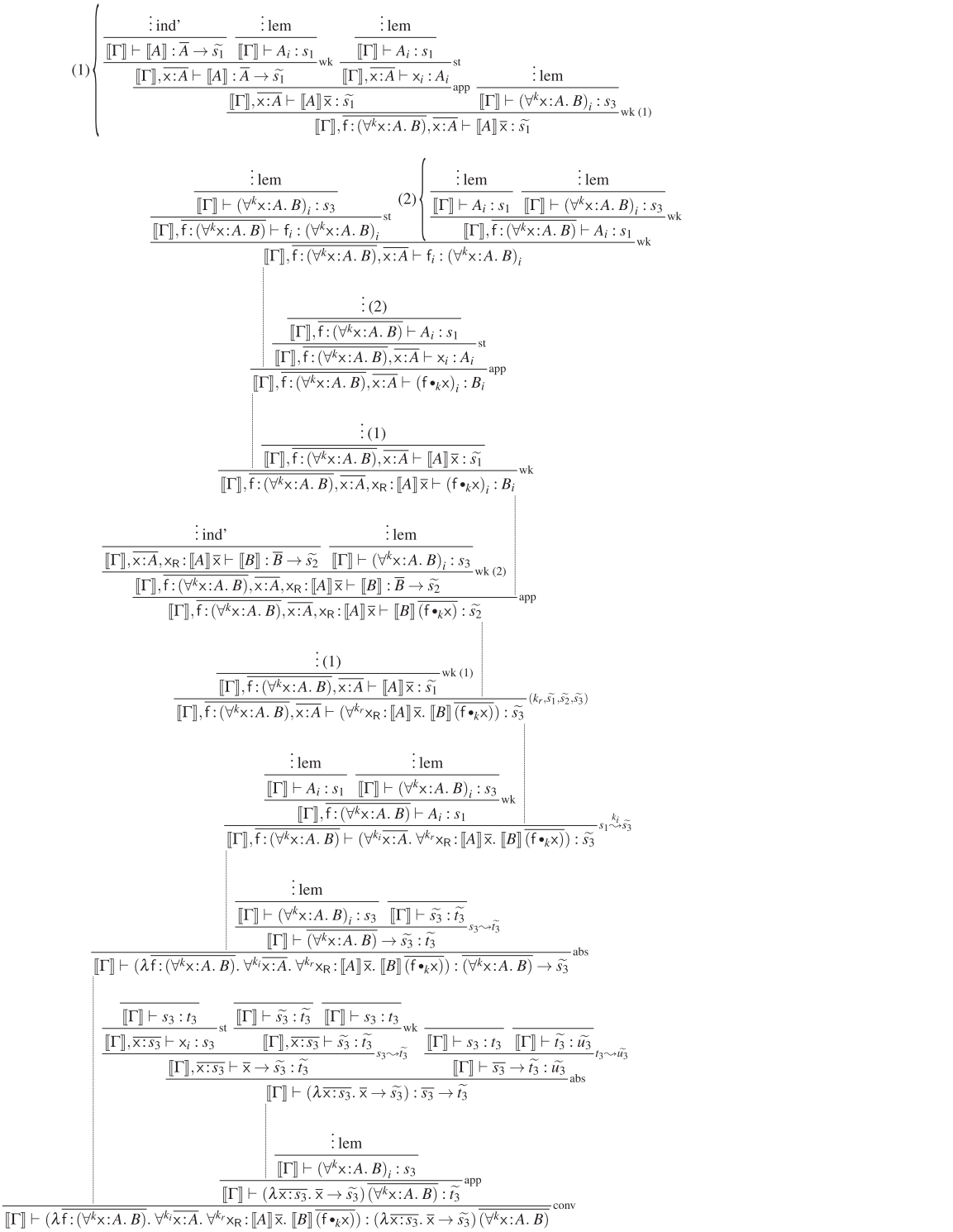
start

$$\dfrac{\dfrac{\dfrac{\vdots\,\text{ind'}}{[\![\Gamma]\!] \vdash [\![A]\!] : \overline{A} \to \widetilde{s}}\quad \dfrac{\vdots\,\text{lem}}{[\![\Gamma]\!] \vdash [\![A_i]\!] : s}}{[\![\Gamma]\!], \overline{\mathsf{x}:A} \vdash [\![A]\!] : \overline{A} \to \widetilde{s}}\,\text{wk} \quad \dfrac{\dfrac{\vdots\,\text{lem}}{[\![\Gamma]\!] \vdash [\![A_i]\!] : s}}{[\![\Gamma]\!], \overline{\mathsf{x}:A} \vdash \mathsf{x}_i : A_i}\,\text{st}}{[\![\Gamma]\!], \overline{\mathsf{x}:A} \vdash [\![A]\!]\,\overline{\mathsf{x}} : \widetilde{s}}\,\text{app}}{[\![\Gamma]\!], \overline{\mathsf{x}:A}, \mathsf{x}_R : [\![A]\!]\,\overline{\mathsf{x}} \vdash \mathsf{x}_R : [\![A]\!]\,\overline{\mathsf{x}}}\,\text{st}$$

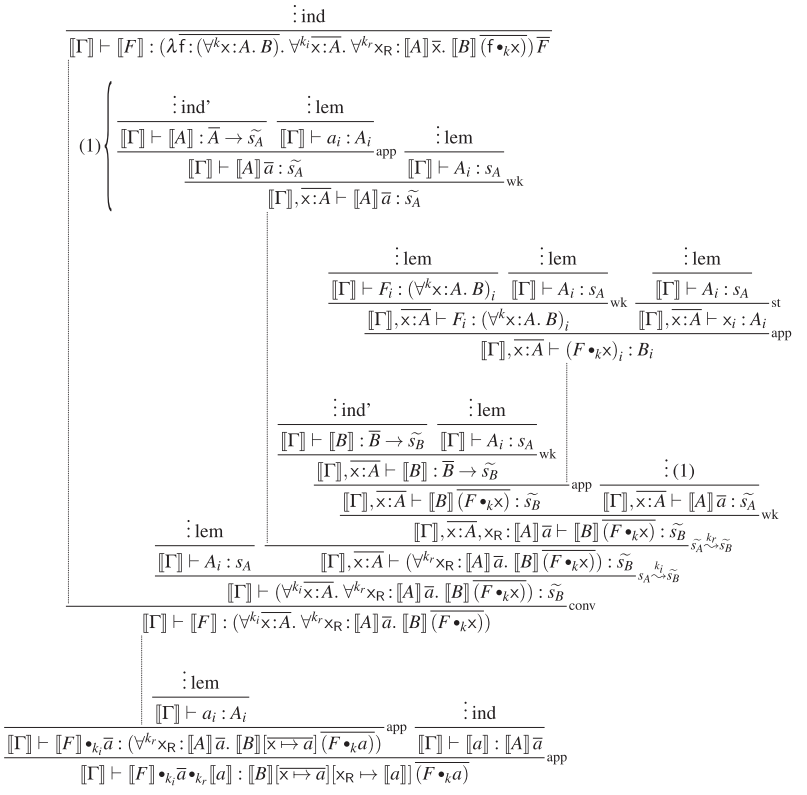
weakening

$$\dfrac{\dfrac{\dfrac{\vdots\,\text{ind}}{[\![\Gamma]\!] \vdash [\![A]\!] : [\![B]\!]\,\overline{A}}\quad \dfrac{\vdots\,\text{lem}}{[\![\Gamma]\!] \vdash C_i : s}}{[\![\Gamma]\!], \overline{\mathsf{x}:C} \vdash [\![A]\!] : [\![B]\!]\,\overline{A}}\,\text{wk} \quad \dfrac{\dfrac{\dfrac{\vdots\,\text{ind'}}{[\![\Gamma]\!] \vdash [\![C]\!] : \overline{C} \to \widetilde{s}}\quad \dfrac{\vdots\,\text{lem}}{[\![\Gamma]\!] \vdash C_i : s}}{[\![\Gamma]\!], \overline{\mathsf{x}:C} \vdash [\![C]\!] : \overline{C} \to \widetilde{s}}\,\text{wk} \quad \dfrac{\dfrac{\vdots\,\text{lem}}{[\![\Gamma]\!] \vdash C_i : s}}{[\![\Gamma]\!], \overline{\mathsf{x}:C} \vdash \mathsf{x}_i : C_i}\,\text{st}}{[\![\Gamma]\!], \overline{\mathsf{x}:C} \vdash [\![C]\!]\,\overline{\mathsf{x}} : \widetilde{s}}\,\text{app}}{[\![\Gamma]\!], \overline{\mathsf{x}:C}, \mathsf{x}_R : [\![C]\!]\,\overline{\mathsf{x}} \vdash [\![A]\!] : [\![B]\!]\,\overline{A}}\,\text{wk}$$

product (k, s_1, s_2, s_3)



application



abstraction We apply the generation lemma (Barendregt 1992, Theorem 5.2.13, case 3) on $\Gamma \vdash (\forall^k \mathsf{x} : A.\ B) : s$. We get: $\exists s_A \rightsquigarrow s_B$ such that

- $\Gamma \vdash A : s_A$
- $\Gamma, \mathsf{x} : A \vdash B : s_B$
- $s =_\beta s_B$

Since sorts are irreducible, the last equation becomes $s = s_B$, so we have: $\exists s_A \rightsquigarrow s$ such that

- $\Gamma \vdash A : s_A$
- $\Gamma, \mathsf{x} : A \vdash B : s$

Induction on the judgements constructed above is valid because the generation lemma generates smaller judgements. It yields:

- $[\![\Gamma]\!] \vdash [\![A]\!] : [\![s_A]\!]\ \overline{A}$
- $[\![\Gamma]\!], \overline{\mathsf{x}:A}, \mathsf{x}_\mathsf{R} : [\![A]\!]\ \overline{\mathsf{x}} \vdash [\![B]\!] : [\![s]\!]\ \overline{B}$,

and these judgements will be used in the construction of the target derivation.

First we show that the type is properly sorted:

$$
\text{(S)}\left\{\;
\dfrac{
\dfrac{\dfrac{\vdots\,\text{lem}}{[\![\Gamma]\!] \vdash (\forall^k \mathsf{x}{:}A.\,B)_i : s}}{[\![\Gamma]\!], \overline{\mathsf{f}:(\forall^k \mathsf{x}{:}A.\,B)} \vdash \mathsf{f}_i : (\forall^k \mathsf{x}{:}A.\,B)_i}\,\text{st}
\quad
\dfrac{\vdots\,(4)}{[\![\Gamma]\!], \overline{\mathsf{f}:(\forall^k \mathsf{x}{:}A.\,B)} \vdash A_i : s_A}\,\text{wk}
}{[\![\Gamma]\!], \overline{\mathsf{f}:(\forall^k \mathsf{x}{:}A.\,B)}, \overline{\mathsf{x}:A} \vdash \mathsf{f}_i : (\forall^k \mathsf{x}{:}A.\,B)_i}
\qquad
\dfrac{\vdots\,(4)}{[\![\Gamma]\!], \overline{\mathsf{f}:(\forall^k \mathsf{x}{:}A.\,B)} \vdash A_i : s_A}\,\text{st}
}{[\![\Gamma]\!], \overline{\mathsf{f}:(\forall^k \mathsf{x}{:}A.\,B)}, \overline{\mathsf{x}:A} \vdash \mathsf{x}_i : A_i}\,\text{app}
}{[\![\Gamma]\!], \overline{\mathsf{f}:(\forall^k \mathsf{x}{:}A.\,B)}, \overline{\mathsf{x}:A} \vdash (\mathsf{f} \bullet_k \mathsf{x})_i : B_i}
$$

$$
\dfrac{
\dfrac{\vdots\,\text{ind'}}{[\![\Gamma]\!] \vdash [\![B]\!] : \overline{B} \to \widetilde{s}}
\quad
\dfrac{\vdots\,\text{lem}}{[\![\Gamma]\!] \vdash (\forall^k \mathsf{x}{:}A.\,B)_i : \widetilde{s}}\,\text{wk}
}{[\![\Gamma]\!], \overline{\mathsf{f}:(\forall^k \mathsf{x}{:}A.\,B)} \vdash [\![B]\!] : \overline{B} \to \widetilde{s}}\,\text{wk}
\qquad
(4)\left\{\;
\dfrac{
\dfrac{\vdots\,\text{lem}}{[\![\Gamma]\!] \vdash A_i : s_A}
\quad
\dfrac{\vdots\,\text{lem}}{[\![\Gamma]\!] \vdash (\forall^k \mathsf{x}{:}A.\,B)_i : s}\,\text{wk}
}{[\![\Gamma]\!], \overline{\mathsf{f}:(\forall^k \mathsf{x}{:}A.\,B)} \vdash A_i : s_A}\,\text{wk}
\right.
$$

$$
\dfrac{
[\![\Gamma]\!], \overline{\mathsf{f}:(\forall^k \mathsf{x}{:}A.\,B)}, \overline{\mathsf{x}:A} \vdash [\![B]\!] : \overline{B} \to \widetilde{s}
}{
[\![\Gamma]\!], \overline{\mathsf{f}:(\forall^k \mathsf{x}{:}A.\,B)}, \overline{\mathsf{x}:A} \vdash [\![B]\!]\ \overline{(\mathsf{f} \bullet_k \mathsf{x})} : \widetilde{s}
}\,\text{app}
$$

$$
\dfrac{
\dfrac{\vdots\,(2)}{[\![\Gamma]\!], \overline{\mathsf{f}:(\forall^k \mathsf{x}{:}A.\,B)}, \overline{\mathsf{x}:A} \vdash [\![A]\!]\ \overline{\mathsf{x}} : \widetilde{s_A}}\,\text{wk}
}{
[\![\Gamma]\!], \overline{\mathsf{f}:(\forall^k \mathsf{x}{:}A.\,B)}, \overline{\mathsf{x}:A}, \mathsf{x}_\mathsf{R} : [\![A]\!]\ \overline{\mathsf{x}} \vdash [\![B]\!]\ \overline{(\mathsf{f} \bullet_k \mathsf{x})} : \widetilde{s}
}
$$

$$
(2)\left\{\;
\dfrac{
\dfrac{\vdots\,(1)}{[\![\Gamma]\!], \overline{\mathsf{x}:A} \vdash [\![A]\!]\ \overline{\mathsf{x}} : \widetilde{s_A}}
\quad
\dfrac{\vdots\,\text{lem}}{[\![\Gamma]\!] \vdash (\forall^k \mathsf{x}{:}A.\,B)_i : s}\,\text{wk (1)}
}{[\![\Gamma]\!], \overline{\mathsf{f}:(\forall^k \mathsf{x}{:}A.\,B)}, \overline{\mathsf{x}:A} \vdash [\![A]\!]\ \overline{\mathsf{x}} : \widetilde{s_A}}
\right.
$$

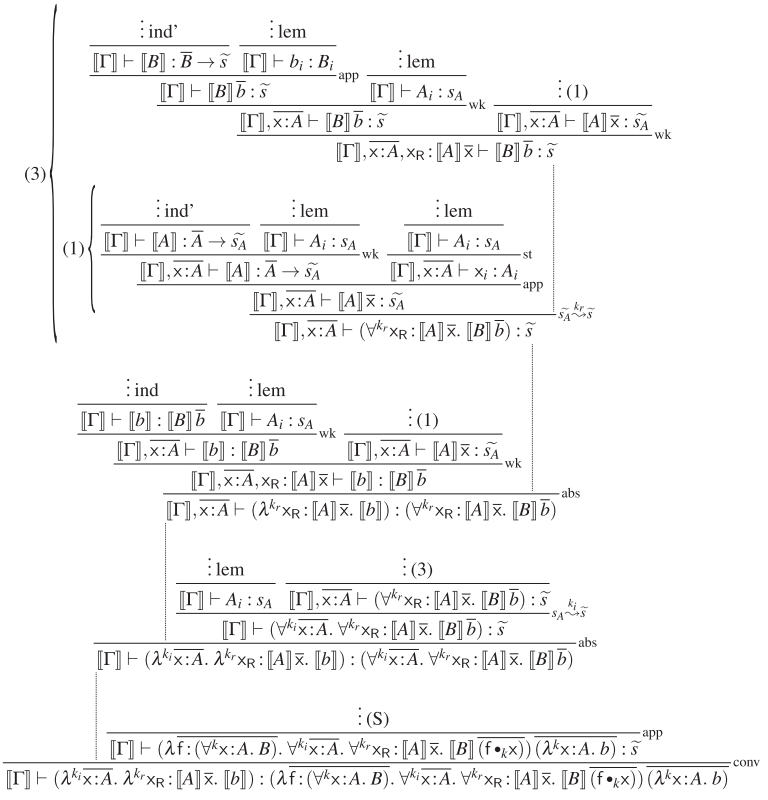
$$
\dfrac{
[\![\Gamma]\!], \overline{\mathsf{f}:(\forall^k \mathsf{x}{:}A.\,B)}, \overline{\mathsf{x}:A} \vdash (\forall^{k_r} \mathsf{x}_\mathsf{R} : [\![A]\!]\ \overline{\mathsf{x}}.\ [\![B]\!]\ \overline{(\mathsf{f} \bullet_k \mathsf{x})}) : \widetilde{s}
}{
[\![\Gamma]\!], \overline{\mathsf{f}:(\forall^k \mathsf{x}{:}A.\,B)} \vdash (\forall^{k_i} \overline{\mathsf{x}:A}.\ \forall^{k_r} \mathsf{x}_\mathsf{R} : [\![A]\!]\ \overline{\mathsf{x}}.\ [\![B]\!]\ \overline{(\mathsf{f} \bullet_k \mathsf{x})}) : \widetilde{s}
}\,{}^{k_r}_{\widetilde{s_A} \rightsquigarrow \widetilde{s}}
$$

$$
\dfrac{
\dfrac{\vdots\,\text{lem}}{[\![\Gamma]\!] \vdash A_i : s_A}
\quad
\dfrac{\vdots\,\text{lem}}{[\![\Gamma]\!] \vdash (\forall^k \mathsf{x}{:}A.\,B)_i : s}\,\text{wk}
}{[\![\Gamma]\!], \overline{\mathsf{f}:(\forall^k \mathsf{x}{:}A.\,B)} \vdash A_i : s_A}\,\text{wk}
}{
[\![\Gamma]\!], \overline{\mathsf{f}:(\forall^k \mathsf{x}{:}A.\,B)} \vdash (\forall^{k_i} \overline{\mathsf{x}:A}.\ \forall^{k_r} \mathsf{x}_\mathsf{R} : [\![A]\!]\ \overline{\mathsf{x}}.\ [\![B]\!]\ \overline{(\mathsf{f} \bullet_k \mathsf{x})}) : \widetilde{s}
}\,{}^{k_i}_{s_A \rightsquigarrow \widetilde{s}}
$$

$$
\dfrac{
\dfrac{
\dfrac{\vdots\,\text{lem}}{[\![\Gamma]\!] \vdash (\forall^k \mathsf{x}{:}A.\,B)_i : s}
\quad
\dfrac{\vdots\,\text{lem}}{[\![\Gamma]\!] \vdash s : \widetilde{t}}
\quad
\dfrac{\vdots\,\text{lem}}{[\![\Gamma]\!] \vdash (\forall^k \mathsf{x}{:}A.\,B)_i : s}\,\text{wk}
}{[\![\Gamma]\!], \overline{\mathsf{f}:(\forall^k \mathsf{x}{:}A.\,B)} \vdash \widetilde{s} : \widetilde{t}}
}{
[\![\Gamma]\!] \vdash (\forall \mathsf{f}:(\forall^k \mathsf{x}{:}A.\,B).\ \widetilde{s}) : \widetilde{t}
}\,{}^{s \rightsquigarrow \widetilde{t}}
$$

$$
\dfrac{
\dfrac{\vdots\,\text{lem}}{[\![\Gamma]\!] \vdash (\lambda^k \mathsf{x}{:}A.\,b)_i : (\forall^k \mathsf{x}{:}A.\,B)_i}
}{
[\![\Gamma]\!] \vdash (\lambda \overline{\mathsf{f}:(\forall^k \mathsf{x}{:}A.\,B)}.\ \forall^{k_i} \overline{\mathsf{x}:A}.\ \forall^{k_r} \mathsf{x}_\mathsf{R} : [\![A]\!]\ \overline{\mathsf{x}}.\ [\![B]\!]\ \overline{(\mathsf{f} \bullet_k \mathsf{x})})\ \overline{(\lambda^k \mathsf{x}{:}A.\,b)} : \widetilde{s}
}\,\text{app}
$$

This sub-proof is then used in the second application of the abstraction rule in the top-level tree.



conversion

$$\dfrac{\dfrac{\vdots\ \text{ind}}{[\![\Gamma]\!] \vdash [\![A]\!] : [\![B]\!]\,\overline{A}} \qquad \dfrac{\dfrac{\vdots\ \text{ind'}}{[\![\Gamma]\!] \vdash [\![B]\!]' : \overline{B'} \to \widetilde{s}} \quad \dfrac{\vdots\ \text{lem}}{[\![\Gamma]\!] \vdash A_i : B'_i}\,\text{app}}{[\![\Gamma]\!] \vdash [\![B]\!]'\,\overline{A} : \widetilde{s}}\,\text{conv}}{[\![\Gamma]\!] \vdash [\![A]\!] : [\![B]\!]'\,\overline{A}}$$

The β-equality constraint ($[\![B]\!]\ \overline{A} =_\beta [\![B']\!]\ \overline{A}$) holds because $[\![\_]\!]$ preserves β-equivalence (Lemma 3.11).

Acknowledgments

Thanks to Andreas Abel, Thierry Coquand, Nils Anders Danielsson, Peter Dybjer, Marc Lasson, Guilhem Moulin, Ulf Norell, Nicolas Pouillard, Janis Voigtländer, Stephanie Weirich, and anonymous reviewers for providing us with very valuable feedback.

References

Abadi, M., Cardelli, L. & Curien, P. (1993) Formal parametric polymorphism. In *Proceedings of POPL'93* (*Charleston, SC*). New York: ACM, pp. 157–170.

Barendregt., H. P. (1992) Lambda calculi with types. *Handbook Log. Comput. Sci.* **2**, 117–309.

Bernardy, J.-P. (2010) Lightweight free theorems: Agda library. Accessed March 13, 2012. Available at: http://wiki.portal.chalmers.se/agda/agda.php?n=Libraries.LightweightFreeTheorems

Bernardy, J.-P., Jansson, P. & Paterson, R. (2010) Parametricity and dependent types. In *Proceedings of ICFP 2010* (*Baltimore, MD*, *September 27–29*). New York: ACM, pp. 345–356.

Bernardy, J.-P. & Lasson, M. (2011) Realizability and parametricity in pure type systems. In *the Proceedings of FoSSaCS 2011* (Saarbruecken, Germany, March 26–April 3), Hofmann, M. (ed), LNCS, vol. 6604. Berlin, Germany: Springer-Verlag, pp. 108–122.

Böhm, C. & Berarducci, A. (1985) Automatic synthesis of typed lambda-programs on term algebras. *Theor. Comp. Sci.* **39**(2–3), 135–154.

Böhme, S. (2007) *Free Theorems for Sublanguages of Haskell*. Master's thesis, Technische Universität Dresden, Netherlands.

Church, A. (1940) A formulation of the simple theory of types. *J. Symb. Log.* **5**(2), 56–68.

Coquand, T. (1986) An analysis of Girard's paradox. In *Logic in Computer Science*, Meyer, A. R. & Chandra, A. K. (eds), Piscataway, NJ: IEEE, pp. 227–236.

Coquand, T. (1992) Pattern matching with dependent types. In *the Proceedings of the Workshop on Types for Proofs and Programs* (*Torino, Italy*), pp. 66–79.

Dybjer, P. (1994) Inductive families. *Form. Asp. Comput.* **6**(4), 440–465.

Gibbons, J. & Paterson, R. (2009) Parametric data-type genericity. In *the Proceedings of WGP 2009* (*Edinburgh, UK, August 30*) New York: ACM, pp. 85–93.

Girard, J.-Y. (1972) *Interprétation Fonctionnelle et Elimination Des Coupures de L'arithmétique D'ordre Supérieur*, Thèeses d'état, Université de Paris, Paris, France.

Hofmann, M. & Streicher, T. (1996) The groupoid interpretation of type theory. In *Venice Festschrift*, Sambin, G. & Smith, J. (eds), Oxford, UK: Oxford University Press, pp. 83–111.

Johann, P. & Voigtländer, J. (2006) The impact of seq on free theorems-based program transformations. *Fundam. Inf.* **69**(1–2), 63–102.

Mairson, H. (1991) Outline of a proof theory of parametricity. In *the Proceedings of FPCA 1991* (*Cambridge, MA, August 26–30*), LNCS, vol. 523. New York: Springer-Verlag, pp. 313–327.

McBride, C. & McKinna, J. (2004) The view from the left. *J. Funct. Program.* **14**(01), 69–111.

Miquel, A. (2001) *Le Calcul des Constructions Implicite: Syntaxe et Sémantique*. Theèses de Doctorat, Université Paris, Paris, France.

Monnier, S. & Haguenauer, D. (2010) Singleton types here, singleton types there, singleton types everywhere. In *the Proceedings of PLPV 2010* (*Madrid, Spain, January 19*). New York: ACM, pp. 1–8.

Morris, P. & Altenkirch, T. (2009) Indexed containers. In *the Proceedings of the Twenty-Fourth IEEE Symposium on Logic in Computer Science*. Piscataway, NJ: IEEE, pp. 277–285.

Neis, G., Dreyer, D. & Rossberg, A. (2009) Non-parametric parametricity. In *Proceedings of ICFP 2009* (*Los Angeles, August*). New York: ACM, pp. 135–148.

Norell, U. (2007) *Towards a Practical Programming Language Based on Dependent Type Theory*. PhD thesis, Chalmers Tekniska Högskola, Gothenburg, Sweden.

Oury, N. & Swierstra, W. (2008) The power of Pi. In *the Proceedings of ICFP 2008* (*Victoria, BC, Canada, September 20–28*). New York: ACM, pp. 39–50.

Paulin-Mohring, C. (1993) Inductive definitions in the system Coq – rules and properties. In *Typed Lambda Calculi and Applications*, Bezem, M. & Groote, J. F. (eds), Berlin, Germany: Springer pp. 328–345.

Plotkin, G. & Abadi, M. (1993) A logic for parametric polymorphism. In *the Proceedings of TLCA '93*, (*Utrecht, The Netherlands, March 16–18*), LNCS, vol. 664. Berlin, Germany: Springer, pp. 361–375.

Pouillard, N. (2011) Nameless, painless. In *the Proceedings of the 16th ACM SIGPLAN International Conference on Functional Programming, ICFP '11* (*Tokyo, Japan, September 19–21*). New York: ACM, pp. 320–332.

Reynolds, J. C. (1983) Types, abstraction and parametric polymorphism. *Inf. Process.* **83**(1), 513–523.

The Coq Development Team. (2010) *The Coq Proof Assistant*. Reference manual. Available at: http://www.coq.inria.fr/doc

Voigtländer, J. (2009) Free theorems involving type constructor classes: Funct. pearl. In *the Proceedings of the 14th ACM SIGPLAN International Conference on Functional Programming* (*Edinburgh, UK, August 31–September 2*). New York: ACM, pp. 173–184.

Vytiniotis, D. & Weirich, S. (2010) Parametricity, type equality, and higher-order polymorphism. *J. Funct. Program.* **20**(02), 175–210.

Wadler, P. (1989) Theorems for free! In *the Proceedings of FPCA 1989* (*London, UK, September 11–13*). New York: ACM, pp. 347–359.

Wadler, P. (2007) The Girard–Reynolds isomorphism. *Theor. Comp. Sci.* **375**(1–3), 201–226.

Wadler, P. & Blott, S. (1989) How to make ad-hoc polymorphism less ad hoc. In *the Proceedings of POPL'89* (*Austin, TX, January 11–13*). New York: ACM, pp. 60–76.